GENERALISED ARC CONSISTENCY FOR THE ALLDIFFERENT CONSTRAINT: AN EMPIRICAL SURVEY

IAN P. GENT, IAN MIGUEL, AND PETER NIGHTINGALE

ABSTRACT. The AllDifferent constraint is a crucial component of any constraint toolkit, language or solver, since it is very widely used in a variety of constraint models. The literature contains many different versions of this constraint, which trade strength of inference against computational cost. In this paper, we focus on the highest strength of inference, enforcing a property known as generalised arc consistency (GAC). This work is an analytical survey of optimizations of the main algorithm for GAC for the AllDifferent constraint. We evaluate empirically a number of key techniques from the literature. We also report important implementation details of those techniques, which have often not been described in published papers. We pay particular attention to improving incrementality by exploiting the strongly-connected components discovered during the standard propagation process, since this has not been detailed before. Our empirical work represents by far the most extensive set of experiments on variants of GAC algorithms for AllDifferent. Overall, the best combination of optimizations gives a mean speedup of 168 times over the same implementation without the optimizations.

1. INTRODUCTION

Constraints are a powerful and natural means of knowledge representation and inference in many areas of industry and academia. Consider, for example, the production of a university timetable. This problem's constraints include: the maths lecture theatre has a capacity of 100 students; art history lectures require a venue with a slide projector; no student can attend two lectures simultaneously. Constraint solving of a combinatorial problem proceeds in two phases. First, the problem is *modelled* as a set of *decision variables*, and a set of *constraints* on those variables that a solution must satisfy. A decision variable represents a choice that must be made in order to solve the problem. The *domain* of potential values associated with each decision variable corresponds to the options for that choice. In our example one might have two decision variables per lecture, representing the time and the venue. For each class of students, the time variables of the lectures they attend may have an AllDifferent constraint on them to ensure that the class is not timetabled to be in two places at once. The second phase consists of using a constraint solver to search for *solutions*: assignments of values to decision variables satisfying all constraints. The simplicity and generality of this approach is fundamental to the successful application of constraint solving to a wide variety of disciplines such as scheduling, industrial design and combinatorial mathematics [29].

The AllDifferent constraint expresses that a vector of variables take distinct values. It is a crucial component of any constraint system, since it is very widely used in a variety of constraint models, for diverse problems such as quasigroup construction and completion, sports scheduling, timetabling and golomb ruler construction. The literature contains many different versions of this constraint, which trade strength of inference against computational cost. Indeed, choosing an appropriate level of consistency is sometimes vital to solving a CSP efficiently [26]. Van Hoeve surveys various strengths of inference [28],

including the weak and fast pairwise decomposition described in Section 6.1.2, bound and range consistency, and generalised arc consistency (GAC). The classic GAC algorithm for the AllDifferent constraint is given by Régin [21]. In this paper, we focus on the highest strength of inference (enforcing GAC). Limiting ourselves to GAC allows us to study various optimizations in great depth, but does mean that the scope of the paper does not include optimizations of bounds and range consistency algorithms.

This work is an analytical survey of optimizations of the main algorithm for generalised arc consistency (GAC) for the AllDifferent constraint. While based on a survey of published optimizations, we extend the literature in three ways. First, we provide extensive implementation details of the optimizations we cover. Such details are often omitted from initial publications, for example for reasons of space. Providing such details should save future workers from having to reinvent the wheel. Second, we provide extensive empirical analyses to show the value or otherwise of the techniques we survey. Importantly, we are able to evaluate optimizations in combination with each other. Third, we go into particular detail on techniques which have not been described in the literature before, and some new techniques we introduce here. In the former category, we show how to explore incrementality in strongly connected components during search. In the latter category, we introduce methods for reducing the number of necessary propagations using dynamic triggers, and an optimization for the case where variables are assigned.

In Section 2 of this paper, we review key background material, present Régin's algorithm at a high level, and survey optimizations for it that have been proposed in the literature. We discuss incremental matching, domain counting, the use of a priority queue, staged propagation, processing strongly connected components independently, important edges, advisors, and fixpoint reasoning. In Section 3 we give implementation details for the algorithm. The detailed survey of key implementation issues is one of the contributions of this paper.

In Section 4, we give extensive details of how to exploit strongly connected components (SCCs) to improve efficiency in AllDifferent propagation. In Régin's algorithm, we find a set of SCCs in a graph formed from allowed values and a matching between variables and values. Edges between distinct SCCs represent impossible values. As we move down a branch in the search tree, an SCC can only remain the same or split into new SCCs. Thus, when we remove a variable-value pair, we need only study the individual SCC which contained the deleted variable-value pair. Since this may contain only a small fraction of all the variables in the original constraint, we can greatly reduce the amount of work that this incremental propagation requires. This paper presents this technique in detail for the first time, since it has previously been in the folklore rather than the literature. We show empirically that this is a very valuable optimization in Section 6. We also give a further, minor, optimization to the computation of SCCs, for the common case that we have assigned a variable to a value.

In Section 5, we show how to exploit "dynamic triggers" for GAC AllDifferent. In a standard algorithm we have to do some work when any value is deleted. Katriel proved that there is a small set of variable-value pairs such that no propagation is possible if any *other* value is deleted [16] and gave a probabilistic algorithm to exploit the idea. We extend this idea to a technique which maintains GAC deterministically, while reducing the number of times the propagator is called. This can be implemented in Minion using dynamic triggers [10]. We show in fact marginally better performance on a version where we implement dynamic triggers *internally* within the AllDifferent propagator. This has the added advantage that it is portable to solvers which do not have a dynamic trigger

infrastructure. We give full details in Section 5 with experimental results in Section 6.4. We conclude that the technique is of benefit mainly where GAC AllDifferent performs badly, so it may not be generally useful.

Our empirical work in Section 6 represents by far the most extensive set of experiments on variants of GAC algorithms for AllDifferent. We implement most (although not all) of the optimizations we have surveyed. Implementations are based on the state-of-the-art Minion constraint solver [9]. Our comparisons are never with a straw-man implementation, as all techniques use the *same* implementation except for the addition of optimizations or the replacement of one technique with another. Our results make a number of points. In some cases, we confirm standard advice from the literature on how to implement GAC AllDifferent. We show in Section 6.3 that incremental matching can reduce runtime, and that as an expensive constraint the AllDifferent constraint should be propagated in a separate queue after cheaper constraints. Also, we show that it is worthwhile to combine the GAC AllDifferent algorithm and a cheaper algorithm into a hybrid *staged* propagator. Against existing advice from the literature, we show that a simpler matching algorithm can be more effective than a more complex one. And, although not reported in the literature before, we show that exploiting strongly connected components is particularly beneficial, with an average speedup of about 3 times. We summarise our advice to future implementers in Section 7.

Compared with a vanilla implementation of Régin's algorithm, our best combination of techniques is always better than not using them, and can speedup search by thousands of times. We get a mean speedup of 168 times over the vanilla implementation. We hope that our survey will help other implementers to obtain these speedups in their systems, and stimulate researchers to invent even more effective optimizations.

2. BACKGROUND

2.1. **Preliminaries.** A CSP $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C} \rangle$ is defined as a set of n variables $\mathscr{X} = \langle x_1, \ldots, x_n \rangle$, a set of domains $\mathscr{D} = \langle D_1, \ldots, D_n \rangle$ where $D_i \subsetneq \mathbb{Z}$, $|D_i| < \infty$ is the finite set of all potential values of x_i, and a conjunction $\mathscr{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_e$ of constraints.

Within CSP $\mathscr{P} = \langle \mathscr{X}, \mathscr{D}, \mathscr{C} \rangle$, a constraint $C_k \in \mathscr{C}$ consists of a sequence of $r > 0$ variables $\mathscr{X}_k = \langle x_{k_1}, \ldots, x_{k_r} \rangle$ with respective domains $\mathscr{D}_k = \langle D_{k_1}, \ldots, D_{k_r} \rangle$ s.t. $\mathscr{X}_k$ is a subsequence[1] of $\mathscr{X}$, $\mathscr{D}_k$ is a subsequence of $\mathscr{D}$, and each variable x_{k_i} and domain D_{k_i} matches a variable x_j and domain D_j in $\mathscr{P}$. C_k has an associated set $C_k^S \subseteq D_{k_1} \times \cdots \times D_{k_r}$ of tuples which specify allowed combinations of values for the variables in $\mathscr{X}_k$.

Although we define a constraint C_k to have scope $\langle x_{k_1}, \ldots, x_{k_r} \rangle$, when discussing a particular constraint we frequently omit the k subscript, and refer to the variables as $\langle x_1, \ldots, x_r \rangle$, and to the domains as $\langle D_1, \ldots, D_r \rangle$.

An AllDifferent constraint is a constraint C_k of any arity, where C_k^S contains all tuples where the values are all distinct. Throughout, we use r as the arity of the AllDifferent constraint in question. We use d to represent the number of domain values involved in the constraint: $d = |D_1 \cup \ldots \cup D_r|$.

A *literal* is defined as a variable-value pair, $x_i \mapsto j$ such that $x_i \in \mathscr{X}$ and $j \in \mathbb{Z}$. To *prune* a literal is to remove the value j from the domain D_i. In the context of a constraint C_k, we refer to a tuple τ of values as being *acceptable* iff $\tau \in C_k^S$, and *valid* iff $|\tau| = r$ and $\forall j : \tau[j] \in D_{k_j}$ (i.e. each value in the tuple is in its respective domain).

[1] We use subsequence in the sense that $\langle 1,3 \rangle$ is a subsequence of $\langle 1,2,3,4 \rangle$.

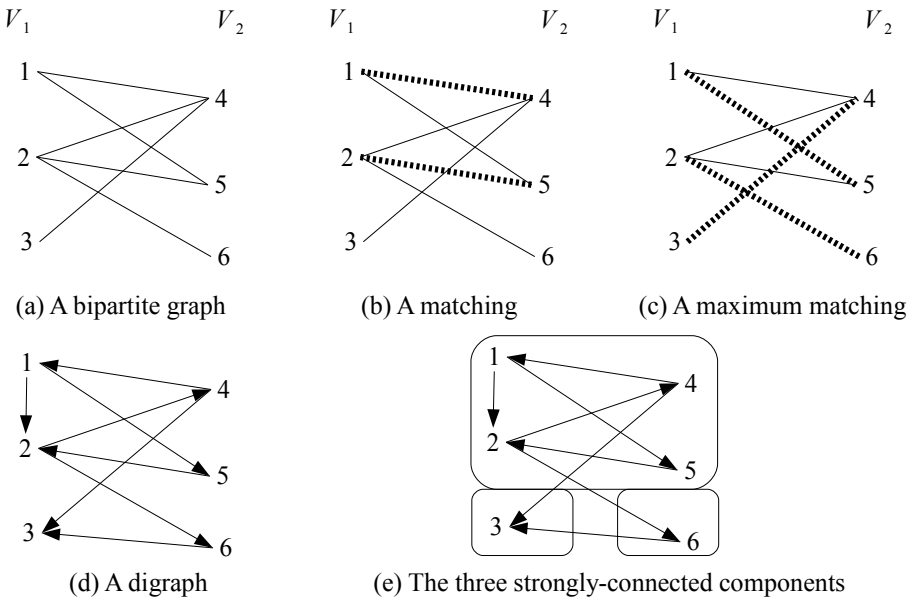


FIGURE 2.1. Examples of graphs

Graph theory. Régin's AllDifferent algorithm [21] makes use of results from graph theory, in particular maximum bipartite matching [3] and strongly connected components [27].

We consider *bipartite graphs* and *digraphs*. A bipartite graph $G = \langle V, E \rangle$ is defined as a set of vertices V and a set of edges $E \subseteq V \times V$, where the edges are interpreted as having no direction, there are no duplicate edges, and the vertices can be partitioned into two sets V_1 and V_2 such that no two elements in the same set are adjacent. Figure 2.1(a) shows an example of a bipartite graph, where $V_1 = \{1,2,3\}$ and $V_2 = \{4,5,6\}$.

A *matching* of a bipartite graph is a set of edges $M \subseteq E$ such that no two edges connect to the same vertex. Figure 2.1(b) shows an example of a matching of cardinality two, where the bold, dotted edges are in the matching. A *maximum matching* (also called a *maximum cardinality matching*) is a matching with the maximum cardinality. Figure 2.1(c) shows a maximum matching for the example bipartite graph. In this case, the maximum matching is unique. There are many algorithms which can compute a maximum matching in a bipartite graph, for example Hopcroft-Karp [14] and Ford-Fulkerson [6].

A digraph is also a pair $G = \langle V, E \rangle$ of a set of vertices V, and a set of edges $E \subseteq V \times V$. Edges are interpreted as having direction. Figure 2.1(d) shows an example of a digraph. A *strongly connected component* (SCC) is a maximal set of vertices of a digraph with the property that there is a path from any vertex to any other in the set. It follows that there are cycles within the SCCs, and no cycles with edges between SCCs. The set of SCCs forms a partition of the vertices of the digraph. Tarjan's algorithm can be used to efficiently compute the SCCs of any digraph in linear time [27]. For figure 2.1(d), the three SCCs are $\{1,2,4,5\}$, $\{3\}$ and $\{6\}$, as shown in figure 2.1(e).

2.2. **Régin and Costa's algorithm.** As pointed out by Knuth and Raghunathan [18], it is well known that the problem of finding a system of "distinct representatives" is equivalent

to bipartite matching. In the context of constraint solving, Régin [21] exploited this equivalence to construct the classic GAC algorithm for the AllDifferent constraint. A very similar algorithm was published by Costa [7] simultaneously. Régin's algorithm has a better time bound. From here on, we will only consider Régin's algorithm.

The algorithm uses results from graph theory, in particular a theorem by Berge [3] (ch. 7, page 125), in an algorithm with two major stages: finding a maximal matching from variables to distinct values, and finding the strongly connected components of a digraph. The algorithm is usually incremental, but for simplicity we summarize it in a non-incremental form:

(1) Find a maximum valid matching M from variables to distinct values. The Hopcroft-Karp [14] or Ford-Fulkerson [6] algorithms may be used for this.
(2) If $|M| < r$, the constraint is unsatisfiable.
(3) Construct the *residual graph* R from M and the variable domains. This digraph has the property that edges between strongly-connected components of R correspond to literals which can be pruned. R is defined below (definition 2.2).
(4) Compute the strongly connected components (SCCs) of R. Tarjan's algorithm [27] may be used for this.
(5) Prune variable-value pairs where the corresponding edge traverses two SCCs in G, and the pair is not contained in M. [2]

2.2.1. *Informal description of the algorithm.* Consider the following example.

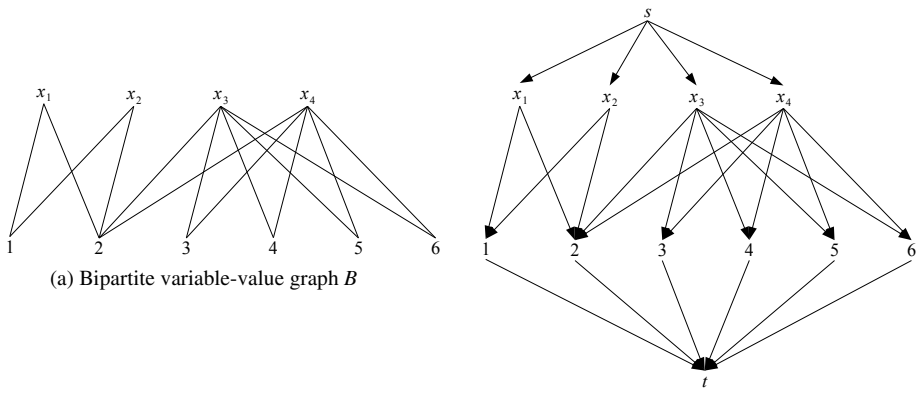
$$x_1, x_2 \in \{1 \dots 2\}, x_3, x_4 \in \{2 \dots 6\} : \text{AllDifferent}([x_1 \dots x_4])$$

Computing the matching. The bipartite variable-value graph is shown in figure 2.2(a). Each variable is represented by a vertex, each value is represented by a vertex, and a variable x_i and value j are connected by an edge iff j is in the domain of x_i. This graph is denoted B. The first stage of the algorithm above is to construct a maximum matching in this graph.

We can consider the bipartite maximum matching problem as a maximum flow problem in a digraph. Maximum flow is the problem of finding the maximum rate at which a material can be shipped from a source vertex to a sink, along the edges in the digraph, without violating capacity constraints on the edges. The digraph is constructed from B by adding a source vertex s and a sink t. The set of edges is as follows: s to all variables $x_1 \dots x_4$, each value $1 \dots 6$ to the sink t, and each edge in B is translated to a directed edge from the variable vertex to the value. This yields the graph in figure 2.2(b), which we refer to as the flow graph. For the purpose of finding a maximum flow, each edge has capacity 1.

The Ford-Fulkerson method [6] (chapter 27) can be used to find a maximum flow in the flow graph from s to t. The algorithm finds an *augmenting path*, which is a path which can be used to increase the flow. For example, $s \rightarrow x_1 \rightarrow 1 \rightarrow t$ is an augmenting path in figure 2.2(b). The algorithm applies the augmenting path to increase the flow, and computes a second flow graph. A second augmenting path is sought in the second flow graph, and in this way Ford-Fulkerson iteratively finds and applies augmenting paths until no such path exists. Hence there is a sequence of flow graphs, culminating in a final flow graph which represents a maximum flow from s to t.

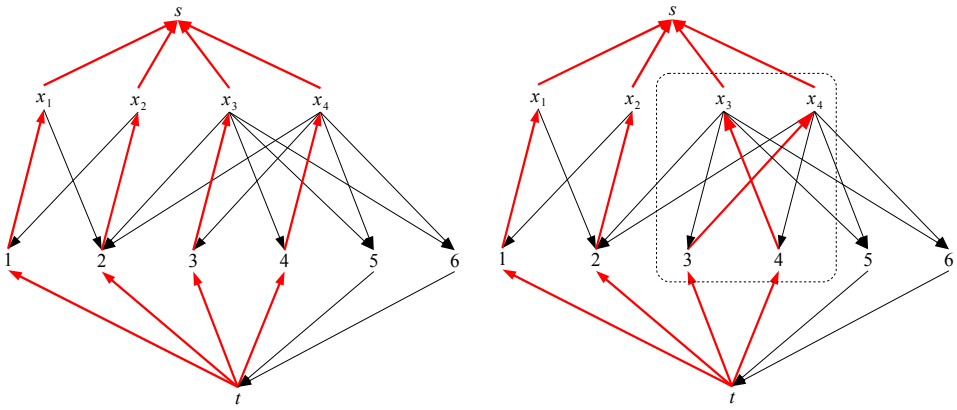
[2] In some cases an edge will be contained in the matching but will also traverse two SCCs. This occurs when an edge is *vital* [21]. For example when a variable is assigned, the variable and the value are each contained in a singleton SCC, but the assignment must be contained in M.



(a) Bipartite variable-value graph B

(b) Flow graph

FIGURE 2.2. Graph for maximum flow algorithm



(a) Final flow graph

(b) Alternate final flow graph

FIGURE 2.3. Two possible final flow graphs

Each edge of any flow graph is either used in the flow from s to t, or unused. An unused edge can carry a new flow of size one, in the direction of the edge in the first flow graph. Conversely, a used edge can carry a new flow in the opposite direction, and become unused. Therefore, in all flow graphs, the direction of used edges is reversed relative to their direction in the first flow graph.

When an augmenting path is applied, each used edge in the path becomes unused and vice versa. This has the effect of reversing the direction of all edges in the path, to create the next flow graph in the sequence.

One possible final flow graph is shown in figure 2.3(a), representing the matching $x_i = i$ for all $i \in 1 \ldots r$. The final flow graph represents a maximum matching, which is used in the next stage of the AllDifferent algorithm.

At this point, the algorithm must test whether the maximum matching covers all variables. If it does not, then the constraint cannot be satisfied, since every variable must be assigned some value (line 2 in the algorithm above).

Computing the SCCs. Consider figure 2.3(a) again. If we take any cycle in this graph, and reverse the direction of all its edges, then we have another flow graph which represents a different maximal matching. For example, if the cycle $t \rightarrow 4 \rightarrow x_4 \rightarrow 5 \rightarrow t$ is reversed, we change the assignment of x_4 to 5 in the matching. Reversing the cycle $3 \rightarrow x_3 \rightarrow 4 \rightarrow x_4 \rightarrow 3$ corresponds to swapping values 3 and 4 in the matching. This is very similar to one iteration of the Ford-Fulkerson algorithm, the difference being that the flow from s to t remains constant, because the path starts and ends at the same vertex. For the cycle $3 \rightarrow x_3 \rightarrow 4 \rightarrow x_4 \rightarrow 3$, the result of reversing the cycle is shown in figure 2.3(b).

It is possible to generate all maximum matchings by finding cycles in the final flow graph, and reversing all edges in the cycle [3] (ch. 7, pages 124-125). In the language of Berge, the cycles which include t correspond to alternating elementary even chains starting at an unsaturated vertex. Cycles which do not include t correspond to alternating elementary cycles.

In figure 2.3(a), the two sets $s_1 = \{x_1, x_2, 1, 2\}$ and $s_2 = \{x_3, x_4, 3, 4, 5, 6, t\}$ are distinct because there is no cycle containing vertices from both sets. In other words, the variables in s_1 cannot be made to take values in s_2 and vice versa. In fact s_1 and s_2 are SCCs (as defined in Section 2.1). Two edges ($x_3 \rightarrow 2$, $x_4 \rightarrow 2$) cross from one SCC to the other.

The final flow graph is partitioned into SCCs. For figure 2.3, the SCCs are $s_1 = \{x_1, x_2, 1, 2\}$, $s_2 = \{x_3, x_4, 3, 4, 5, 6, t\}$, and $s_3 = \{s\}$. For any edge which crosses from one SCC to another, and is not contained in the matching, its corresponding domain value is removed. This is sufficient to enforce GAC [21]. In this example, 2 is removed from the domain of x_3 and x_4 because of the edges $x_3 \rightarrow 2$ and $x_4 \rightarrow 2$.

2.2.2. *The AllDifferent algorithm in detail.* To construct a sound and complete GAC algorithm [21], Régin exploited a result by Berge, applied to B: an edge is *free* (belonging to some, but not all, of the maximum matchings) iff the edge belongs to a path which is *even* (containing an even number of edges) and *alternating* (the path alternates between edges in the matching and not) beginning at an unmatched vertex, or an alternating cycle [3] (ch. 7, page 125). Edges which are neither free nor in the matching are in no maximum matching, and correspond to a variable-value pair that can therefore be pruned. We now describe Régin's algorithm in detail.

In the informal description above, we used the Ford-Fulkerson algorithm to compute the matching, and its final residual graph is used in the second stage of the algorithm. Ford-Fulkerson may not be the most efficient matching algorithm, so in this section we separate the matching from the SCC computation.

The size of the union of all domains is $|D_1 \cup \ldots \cup D_r| = d$. For simplicity, domain elements are assumed to be $1 \ldots d$.

Definition 2.1. The bipartite variable-value graph is defined as $B = \langle V, E \rangle$ where $V = \{x_1, \ldots, x_r, 1, \ldots, d\}$ and $E = \{x_i \leftrightarrow j \,|\, j \in D_i\}$

A bipartite matching algorithm is applied to B, returning a maximum-cardinality matching $M : \{x_1, \ldots, x_r\} \rightarrow \{1, \ldots, d\}$. If $|M| < r$ then the AllDifferent constraint is not satisfiable, and the algorithm returns false.

Next we construct the residual digraph as a function of M and $D_1 \ldots D_r$. This is similar to the final flow graph, except that we omit the source vertex s since it is always a singleton SCC, and the direction of each edge is opposite (for efficiency reasons; this does not affect the SCCs).

Definition 2.2. The residual digraph is defined as $R = \langle V', E' \rangle$ where $V' = \{x_1, \ldots, x_r, 1, \ldots, d, t\}$ and there are four types of edges: $E' = M \cup E_2 \cup E_3 \cup E_4$. The matching edges M connect

variables to values. Residual edges connect values to variables, where the value is not used in the matching: $E_2 = \{j \mapsto x_i \,|\, j \in D_i \wedge (x_i \mapsto j) \notin M\}$. A third set of edges connect to t: $E_3 = \{j \mapsto t \,|\, \exists i : (x_i \mapsto j) \in M\}$ and a fourth set connect from t to unmatched values: $E_4 = \{t \mapsto j \,|\, (\nexists i : (x_i \mapsto j) \in M) \wedge (\exists i : j \in D_i)\}$.

An SCC algorithm is applied to R, returning a partition $S = \{s_1, \ldots, s_k\}$ of V'. If $k = 1$ then the algorithm is finished. Otherwise, for each edge in B which is not contained in M, and which connects vertices in two SCCs in S, the edge corresponds to a variable-value pair which is pruned.

2.3. **Optimizations to the basic algorithm.** A number of optimizations to the basic algorithm have been proposed by various authors. We survey them here.

2.3.1. *Incremental matching.* The matching M may be maintained incrementally during search [21]. Régin suggested that a representation of the variable-value graph and the set of edges involved in the matching would be stored between calls. Deleted edges would be restored upon backtracking beyond the decision that caused their removal. Edges corresponding to pruned values would be removed incrementally. In order to remove the appropriate edges, Régin's algorithm requires that a set of removed values is passed into the propagator.

If Hopcroft-Karp is used to repair the matching, Régin reports that the time complexity of Hopcroft-Karp, and of the AllDifferent algorithm as a whole, is improved from $O(r^{1.5}d)$ to $O(\sqrt{k}rd)$, where k is the number of matching edges which have been lost [21].[3]

2.3.2. *Domain counting.* Quimper and Walsh [20] proposed variants of the AllDifferent and Global Cardinality constraints for set, multiset and tuple variables. These variable types can have extremely large domains. They observe that if the domain of a variable x_i is larger than some threshold, then the constraint need not be triggered by any pruning from x_i. The threshold value is always less than or equal to r. Quimper and Walsh give an algorithm that calculates the size of all domains before constructing a set of variables whose domains are smaller than their threshold value. While this idea was conceived in the context of set, multiset and tuple variables, it may apply to small finite domains as well.

We suspect that counting all domains would be expensive. In our experiments, we follow the simpler approach of Lagerkvist and Schulte [19], which uses r as the threshold value. When a variable event triggers the constraint, the domain D_i of the variable is counted. If $|D_i| \leq r$, then the propagator is called (or queued to be called).

We observe that the threshold can be reduced to $r-1$. The original lemma [20] is based on Hall sets, where a Hall set is a set H of variables such that each variable domain is a subset of a set of values D_H, and $|D_H| = |H|$. If a Hall set exists, then all values in D_H are pruned from all variables not in H. If all Hall sets are found and corresponding pruning is performed, then GAC is established. A Hall set of size r is of no use, because there are no variables outside the Hall set to be pruned. The largest Hall set which is useful is of size $r-1$, where all domains are of size $r-1$ or less. Therefore in our experiments with domain counting, the propagator is called (or queued to be called) only if D_i is changed and $|D_i| < r$.

As an example of a Hall set, consider the following AllDifferent constraint: $x_1 \ldots x_3 \in \{1 \ldots 3\}, x_4 \ldots x_6 \in \{3 \ldots 6\} : \text{AllDifferent}(x_1 \ldots x_6)$. The variables $x_1 \ldots x_3$ form a Hall set,

[3] In a more recent presentation (see http://www-sop.inria.fr/coprin/cpaior04/files/graphandcp.ppt slide 70), Régin revises this bound to $O(krd)$.

with values $1 \ldots 3$. Therefore, value 3 is removed from the domains of variables $x_4 \ldots x_6$. This type of reasoning is used informally to solve Sudoku puzzles.

2.3.3. *Priority queue.* Many constraint solvers have a priority queue for constraints [1, 15, 23], such that the priorities determine the order in which constraint propagators are executed. It is standard practice for the AllDifferent constraint to have a low priority. Schulte and Stuckey demonstrate the importance of priority queueing [22], and we evaluate it in our experiments.

2.3.4. *Staged propagation.* Schulte and Stuckey also propose multiple or staged propagation [22], where a cheap propagator with a high priority is combined with a more expensive, low priority propagator. For instances containing AllDifferent this approach shows some promise, so we evaluate a staged propagator in our experiments.

2.3.5. *Decomposition of AllDifferent.* Suppose for example we have AllDifferent$(x_1 \ldots x_6)$ and have $x_1 \ldots x_3 \in \{1 \ldots 3\}$, $x_4 \ldots x_6 \in \{4 \ldots 6\}$. This can decompose into independent constraints AllDifferent$(x_1 \ldots x_3)$ and AllDifferent$(x_4 \ldots x_6)$. This decomposition saves time if we can efficiently find and manage the decompositions. Unfortunately, this optimization is in the folklore of the community rather than the literature.[4] One cheap method of decomposing the constraint is to use the strongly connected components found by the GAC AllDifferent algorithm. As we report in Section 4, this method is not complete (it does not find all possible decompositions), but it works well in practice. How to manage the decompositions efficiently is less obvious. One of the contributions of this paper is to describe, in Section 4, how to implement this optimization efficiently, including algorithms and data structures.

2.3.6. *Important Edges.* Katriel observed that many value removals affecting an AllDifferent result in no other value removals, and so work processing them is wasted [16]. In the more general context of constraints based on network flows, she introduces the concept of an "important edge". An important edge is one whose removal causes the removal of some variable-value pair. She gives an upper bound for the number of important edge for AllDifferent and generalised cardinality constraints. If the graph is dense enough (i.e. if there are many allowed values per variable), she shows that expected cost of propagation can be reduced by only propagating intermittently. She suggests firing propagation when a simple count of value removals affecting constraints hits a number indicating that, probabilistically, a value is likely to be deleted. Note that this algorithm does not actually enforce GAC, because it may miss propagations when the count is low, catching them at a deeper node in the search tree. Katriel does not report an implementation, and observes that the risks of failing to propagate may outweigh the reduced cost of propagation.

While an implementation of Katriel's algorithm would be interesting, the fact that it does not maintain GAC puts it outside the scope of this paper. In this paper we adapt the notion of important edges to reduce the number of times the AllDifferent propagator is called while still guaranteeing that GAC is enforced correctly. We do this by using "dynamic triggers" [10], and report on this in detail in Section 5. We describe a cheap technique for finding at most $2r + d$ edges such that any edge not in the set is guaranteed not to be an important edge.

[4] To see that the idea is known, see slide 61 of [4], from which our example is taken.

Algorithm 1 AllDifferent most basic variant

propagate(): returns Boolean

(1) for i in $\{1 \dots r\}$: matching[i]$\leftarrow i$ // initialize the matching
(2) hasMatching$\leftarrow$FindMaximumMatching(matching, $\{x_1 \dots x_r\}$) // repair the matching
(3) **if not** hasMatching:
(4) return False
(5) FindSCCsRemoveValues(matching, $\{x_1 \dots x_r\}$)
(6) return True

2.3.7. *Other proposals in the literature.* Lagerqvist and Schulte develop *advisors* in the context of the Gecode solver [19]. An advisor is a procedure which is executed immediately when a variable event occurs. They use advisors with AllDifferent to implement domain counting, and to eagerly maintain the matching whenever it is violated. Unfortunately, for all the AllDifferent variants and problem instances they experimented with, the cost of advisors outweighed their benefits [19]. We did not experiment with advisors.

Schulte and Stuckey propose fixpoint reasoning [22] and observe a very slight (0.1%) improvement in runtime for their instance golomb-10-d which has a GAC AllDifferent constraint. Fixpoint reasoning can reduce the number of calls to the propagator, by eliminating useless calls. However, they observe that useless execution of the AllDifferent algorithm is cheap due to its incrementality. We did not experiment with fixpoint reasoning.

3. Implementation of the AllDifferent algorithm

In this section we report on a number of implementation details of the GAC AllDifferent propagator, which we then use for empirical evaluation of optimizations. This gives a survey of the major and some minor issues facing any implementer. With the exception of two matching algorithms, we make design decisions either based on the literature or based on what seems reasonable for the purpose, rather than subjecting our choices to rigorous empirical evaluation. In this section we do select two bipartite maximum matching algorithms for later empirical comparison, selecting one which has a good time bound and another which is known to work well in practice.

Algorithm 1 shows the most basic variant of the AllDifferent propagator. This variant is not incremental in any way. It simply calls FindMaximumMatching and FindSCCsRemoveValues.

To support incremental matching, line 1 would be removed. The two matching algorithms we consider both perform iterative repair, so no changes need to be made there to support incremental matching. This variant of AllDifferent has one item of state which is stored from one call to the next (the matching function). This is not backtracked, because a valid matching is backtrack stable. As values are restored on backtracking, a valid matching remains valid since none of the values in it are removed.

Both variants of AllDifferent call FindMaximumMatching and FindSCCsRemoveValues. These two functions are described in sections 3.1 and 3.2 below.

Régin claims that the space complexity of AllDifferent is $O(rd)$ because the variable-value graph is stored explicitly [21]. In Régin's approach, the variable-value graph is maintained as values are removed from domains, and it must be backtracked as search backtracks. This could be justified in a context where querying domains is expensive.

However, in our experimental context, querying domains is cheap. Therefore, in our implementation, we do not store either the variable-value graph or the residual graph explicitly, reducing the space complexity to $O(d)$. The graphs are discovered as they are traversed. Since most edges correspond to a variable-value pair, checking if an edge is present is implemented as testing if a domain contains a particular value. When discovering all edges from a variable vertex, it is necessary to iterate over a domain. Ideally the solver would provide a domain iterator, which would find the first and next domain elements in constant time. In our experiments, the solver provides the minimum and maximum values of the domain, which we use to bound the iteration. To find all edges from a value vertex to a variable, it is necessary either to iterate through all variables (as we do herein), or to store and backtrack the graph explicitly.

3.1. **Maximum Bipartite Matching.** The first algorithm we considered was the Hopcroft-Karp algorithm [14]. The Hopcroft-Karp algorithm runs in time $O(\sqrt{r}m)$ where m is the number of edges in the variable-value graph ($m \leq rd$). However (when using incremental matching) the algorithm only computes a matching from scratch at the root node of search; subsequently it repairs a matching where k edges have been lost. With Hopcroft-Karp the cost is reported to be $O(\sqrt{k}m)$ [21]. Our implementation in C++ follows that of Eppstein [8].

The second algorithm we implemented was Ford-Fulkerson [6] with a simple breadth-first search (FF-BFS) for augmenting paths. It begins with an unmatched variable vertex, and searches for an augmenting path. The augmenting path is then applied to increase the cardinality of the matching by one. This is iterated until there are no more unmatched variable vertices, or the BFS does not find an augmenting path.

FF-BFS has the advantage of good average behaviour on a wide range of bipartite graphs [25], although the algorithm runs in time $O(rm)$. To repair a matching where k edges have been lost, the cost is $O(km)$.

Régin [21] used the Alt, Blum, Mehlhorn and Paul [2] (ABMP) algorithm, which is a variant of Hopcroft-Karp with a time bound of $O(r^{1.5}\sqrt{m/\log r})$. In terms of the upper bound, Hopcroft-Karp is better for sparse graphs, whereas ABMP is better for dense graphs. We do not know the density of the variable-value graph in advance.

Compared to other applications of matching algorithms, our graphs are relatively small. In our experiments, the problem class with the largest AllDifferent constraint is the contrived problem ($r = 500$). The second largest is for social golfers ($r = 480$), and the third is sports scheduling ($r = 120$). This is not because our instances are easy; many take over two hours to solve.

Setubal empirically compared ABMP, FF-BFS, FF-DFS (Ford-Fulkerson with depth-first search) and Goldberg's algorithm [25]. He generated bipartite graphs with 2^p vertices in each partition, where $p \in \{8 \ldots 17\}$. If we estimate that our graphs have 2^9 vertices in each partition, an examination of Setubal's results (taking the size closest to 2^9 for each class of graphs, and only considering sequential computers) shows that FF-BFS is competitive for all classes and is most efficient (or equal) in 8/11 classes.

Taking these results together with earlier work by Setubal [24], we expect FF-BFS to perform better than Hopcroft-Karp in our experiments. This is the case, as shown in section 6.3.

3.2. **Finding SCCs and removing domain values.** To compute the SCCs, we use Tarjan's algorithm [27], since it is simple and efficient (with a time bound of $O(|V| + |E|)$

Algorithm 2 FindSCCsRemoveValues

```
FindSCCsRemoveValues(matching, varSet): returns nothing
(1)  visited← ∅; TStack← []; maxDFS← 1; hasSCCSplit←False
(2)  for x_i ∈varSet:
(3)      if x_i ∉visited:
(4)          TarjanRemoveValues(x_i) // start search at x_i
```

```
TarjanRemoveValues(curnode): returns nothing
(1)  TStack.push(curnode)
(2)  DFSNum[curnode]←maxDFS
(3)  lowLink[curnode]←maxDFS
(4)  maxDFS ← maxDFS + 1
(5)  visited.insert(curnode)
(6)  for newnode∈neighbourhood(curnode):
(7)      if newnode∈visited:
(8)          if newnode∈TStack:
(9)              lowLink[curnode]←min(lowLink[curnode], DFSNum[newnode])
(10)     else:
(11)         TarjanRemoveValues(newnode)
(12)         lowLink[curnode]←min(lowLink[curnode], lowLink[newnode])
(13) if lowLink[curnode]=DFSNum[curnode]: // if curnode is the root of an SCC
(14)     if lowLink[curnode]>1 or DFS did not traverse all variables:
(15)         hasSCCSplit←True
(16)     if hasSCCSplit:
(17)         SCC← ∅; stacknode←null
(18)         while stacknode≠curnode:
(19)             stacknode←TStack.pop()
(20)             SCC.insert(stacknode)
(21)         for e ∈SCC where e ∈ {1...d}: // e is a domain value
(22)             for x_i ∈varSet where x_i ∉SCC:
(23)                 removeFromDomain(x_i, e)
```

or $O(rd)$). It is also suitable for the optimization we describe in Section 5, where some information is collected from the algorithm as it runs.

Algorithm 1 calls FindSCCsRemoveValues (algorithm 2), which finds SCCs and removes the appropriate values to achieve GAC. To avoid storing all the SCCs explicitly, these two tasks are implemented together. FindSCCsRemoveValues is a simple wrapper which initializes data structures and calls TarjanRemoveValues, possibly more than once as needed. All variables are shared between FindSCCsRemoveValues and TarjanRemoveValues.

TarjanRemoveValues performs Tarjan's algorithm [27] recursively (lines 1-12), and removes values from domains using the SCCs (lines 13-23). Tarjan's algorithm performs a depth-first search (DFS). If it is implemented recursively (as it is here), the SCCs can be constructed as the recursion unwinds. The crux of Tarjan's algorithm is the *root property*, by which a vertex is identified as the root of an SCC. The root property is tested on line 13. When a root is identified, the SCC is constructed from the TStack data structure.

The residual graph (definition 2.2) is given by the neighbourhood function (line 6), where neighbourhood(v) returns the set of vertices $\{v_1, v_2, \ldots\}$ that are connected to v by a directed edge $v \mapsto v_j$.

The data structures of Tarjan's algorithm are described below.

- **maxDFS** is a simple counter, starting at 1, which is incremented each time a new vertex is discovered (on line 4).
- **DFSNum** is used to record the order in which vertices are discovered, by numbering the vertices from 1. DFSNum is set from maxDFS on line 2.
- **visited** is the set of vertices that have been visited. It is updated on line 5.
- **TStack** is a stack of vertices, which starts empty. Whenever a new vertex is discovered, it is pushed onto TStack (line 1). When the algorithm detects that curnode (the current node in the DFS) is the root of an SCC, it constructs the SCC by popping all values from TStack, up to and including curnode (lines 17-20).
- **lowLink** is the key data structure, with the property that lowLink[c] is the minimum of the DFSNum of all vertices reachable by following edges used by the DFS, followed by at most one other edge in R. For each vertex c, lowLink[c] equals DFSNum[c] iff c is the root of a strongly connected component. lowLink[c] is initialized to DFSNum[c]. The value of lowLink[c] is updated from neighbouring vertices on lines 9 and 12. When lowLink[c] is used on line 13, it has reached its final value.

The neighbourhood function does not appear in the program code, to avoid the overhead of a function call. Instead, lines 6-12 are repeated three times for the following three cases: where curnode=t, curnode$\in \{x_1 \ldots x_r\}$, or curnode$\in \{1 \ldots d\}$. For the case where curnode$= t$, the neighbour set is computed ahead of time. If this set is empty, t is omitted from the residual graph, since it will be a singleton SCC. When curnode$\in \{x_1 \ldots x_r, 1 \ldots d\}$, the neighbour set is iterated without being explicitly constructed beforehand.

When the algorithm finds the root of an SCC (line 13), it does a simple test to determine if the residual graph is partitioned (lines 14-15). If the DFS has not traversed all variable vertices (because some are unreachable), or the recursion did not unwind fully, then the residual graph must partition into more than one SCC.

If the residual graph does partition, the algorithm computes the current SCC (lines 17-20). This SCC contains both variables and values, such that in all maximum matchings, the values are assigned to variables within the SCC. Therefore these values cannot be assigned to any variable from varSet which is not in the SCC. Accordingly, these values are removed from all such variables using removeFromDomain.

3.3. **Minor implementation details.** The graph algorithms manipulate small sets of integers, performing operations such as inserting and removing integers, clearing the set, testing the presence of a particular integer, and iterating through the set. One example is the set of visited vertices in Tarjan's algorithm. (In the implementation, each vertex is mapped to a distinct integer.) In all cases we know the range of the integers (it is never more than $0 \ldots r + d$).

We designed the following data structure to represent a subset of $0 \ldots n$. We have an integer array $v[0 \ldots n]$, indexed by set element, and an integer c (called the certificate). v is initialized to 0, and the certificate to 1. An element e is present in the set iff $v[e] = c$. To insert an element e, $v[e] \leftarrow c$, and to delete e, $v[e] \leftarrow 0$. To clear the set, $c \leftarrow c + 1$. In this way, we can clear the set in small constant time.

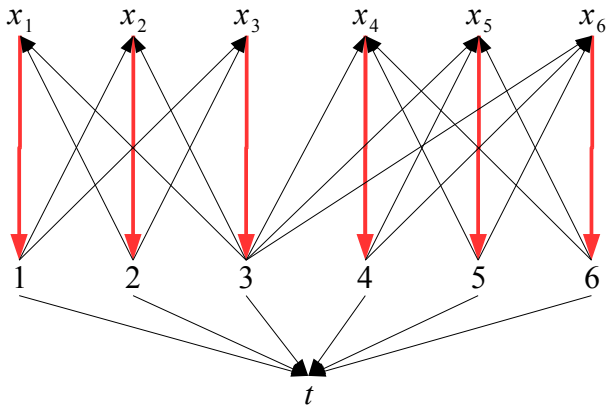


FIGURE 4.1. Residual graph example

When we require a set to be iterable, we also maintain an array of the values, stored contiguously, with an integer representing the size of the set. The clear operation can still be implemented in constant time by setting the size to 0 and incrementing the certificate. For this type of set the remove operation is linear time, but it is very rarely used, being called in only one place. It is called once for each application of the assignment optimization (described in Section 4.2).

The matching M is primarily represented as an array of domain values, indexed by variable number. When using Hopcroft-Karp, it is also represented as an array of variable numbers, indexed by value. Both Hopcroft-Karp and FF-BFS maintain the set of values in the matching. Hopcroft-Karp also maintains the set of variables.

4. EXPLOITING STRONGLY CONNECTED COMPONENTS

In this section we focus on optimizations that exploit strongly connected components (SCCs) during the search process. The main focus is the first detailed description in the literature of how to exploit the independence of strongly connected components, which we described briefly in Section 2.3.5. Here, we describe data structures and algorithms for the independent processing of SCCs. We also introduce what we believe to be a new, but minor, optimization to process SCCs faster when a variable has been assigned.

$$x_1 \dots x_3 \in \{1 \dots 3\}, x_4 \dots x_6 \in \{3 \dots 6\} : \text{AllDifferent}(x_1 \dots x_6) \tag{4.1}$$

To see how we can identify disjoint SCCs, consider (4.1). Assume that the matching found is $x_i \mapsto i$ for all i. The residual graph is shown in figure 4.1. Running Tarjan's algorithm on this graph computes two SCCs containing variables $\{x_1 \dots x_3\}$ and $\{x_4 \dots x_6\}$ respectively. After partitioning the graph, the GAC algorithm would prune value 3 from variables $x_4 \dots x_6$. At this point, the two SCCs are completely disconnected in the residual graph and in the variable-value graph, and will remain so until values are restored by backtracking. In future calls to the propagator, the two SCCs can be considered independently. This allows us to speed up both the maximum matching algorithm and Tarjan's algorithm. The two algorithms are run on a subgraph of the variable-value graph (the maximum matching algorithm), or on a subgraph of the residual graph (Tarjan's). Furthermore, when a variable x_i is changed and the changes trigger the AllDifferent constraint, only the

SCC containing x_i needs to be considered. These changes result in considerable efficiency gains, as shown in Section 6.5 below.

This method does not perform all possible decompositions of the AllDifferent constraint. Suppose for example we have AllDifferent$(x_1 \dots x_6)$ and have $x_1 \dots x_3 \in \{1 \dots 4\}$, $x_4 \dots x_6 \in \{5 \dots 8\}$. This can decompose into independent constraints AllDifferent$(x_1 \dots x_3)$ and AllDifferent$(x_4 \dots x_6)$. Our method would not perform this decomposition. Both parts of the constraint have three variables and four values, and the spare values cause $x_1 \dots x_3$ and $x_4 \dots x_6$ to be connected through the sink t, hence all six variables are contained within one SCC. Despite its incompleteness, decomposing according to SCCs works well in practice, as we will show below.

4.1. **Representing set partition.** In order to store the SCCs between calls to the AllDifferent algorithm, a backtrackable representation of set partition is needed. It is sufficient to store the partition of the set of variables (represented as integers $1 \dots r$), since the values can be quickly discovered from the variables. It is important that each set in the partition is efficiently iterable, since both the matching algorithm and Tarjan's algorithm need to iterate over the set of variables. The order of iteration is not important. It is not necessary to have an $O(1)$ set membership test.

When the AllDifferent algorithm executes, it may subdivide the SCCs further but it never merges SCCs together or changes them in any other way. Therefore only subdivision is required, with the sets being restored on backtracking.

For a set of integers $S = \{1 \dots r\}$, the partition representation we used consists of two arrays of integers, and an array of backtracking Booleans.

setElements[$1 \dots r$]: Contains a permutation of the elements in S.

setElementIndex[$1 \dots r$]: For each element $a \in S$, setElements[setElementIndex[a]]$= a$.

splitPoint[$1 \dots r-1$]: If splitPoint[b]=*False*, elements setElements[b] and setElements[$b+1$] are in the same subset. Otherwise, elements setElements[b] and setElements[$b+1$] belong to different subsets in the partition.

The operation of subdividing the partition involves permuting the elements in setElements (and updating setElementIndex accordingly), and changing Booleans in splitPoint from *false* to *true*. When this change is backtracked, it is only necessary to restore the splitPoint array. This is illustrated in figure 4.2 for a simple example.

To subdivide a subset of size n takes $O(n)$ time, since n elements may need to be written in the setElements array, and n indices updated in the setElementIndex array. Up to $n-1$ elements of splitPoint may be changed in this operation. To undo this operation on backtracking, up to $n-1$ values of splitPoint are restored.

The setElements and setElementIndex arrays are simple arrays of integers. The splitPoint array is maintained by trailing. Changing a value in the splitPoint array has $O(1)$ cost overall. It involves three operations which each take $O(1)$ time: changing the value in memory, adding a record to the trail stack, and also reading the record and restoring the value on backtracking. Therefore the cost of maintaining splitPoint does not affect the overall $O(n)$ time to subdivide the partition.

4.2. **Assignment optimization.** Assignment of a variable, whether by the search procedure or by propagation, is likely to be a common enough case that optimizing it will pay off. When a variable x_i is assigned, the computation of SCCs can be simplified somewhat. In the residual graph, x_i has one outward edge and no inward edges, therefore x_i must be

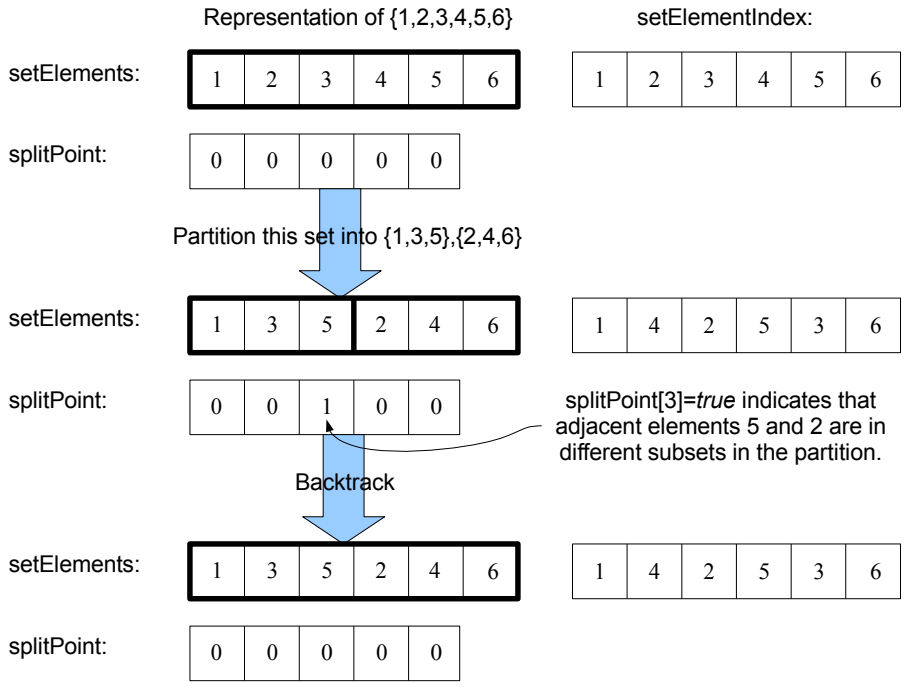


FIGURE 4.2. Illustration of set partition data structure

in a singleton SCC. Where $x_i \mapsto a$, value a must be removed from the domain of all other variables.

To optimize this case, the SCC s containing x_i is partitioned. First x_i is swapped with the first element of the SCC in setElements. Then splitPoint[setElementIndex[i]] is set to *True* to subdivide s into $s_1 = \{x_i\}$ and $s_2 = s \setminus \{x_i\}$. This process is illustrated in figure 4.3. The value a is removed from the domain of all variables in s_2, and s_2 is queued to be processed by Tarjan's algorithm (since it may subdivide further). This takes $O(r)$ time, and does not decrease the number of calls to Tarjan's algorithm. However it does reduce the size of the graph which Tarjan's operates on. The effectiveness of this optimization is tested in Section 6.5.

4.3. **Implementing independent SCCs.** To implement both the above proposals, we replace the simple propagate function (algorithm 1) with propagate-SCC (algorithm 3). This function requires a set of variables named triggeringVars as a parameter. These are the variables which have triggered the constraint: in the simplest case this would be all variables whose domain has changed since the last call to propagate-SCC. When dynamic triggers (described in Section 5) are used, triggeringVars is the set of all variables which have lost one or more of their trigger values.

When domain counting is used, triggeringVars is the set of all changed variables whose domain size is less than r. For a variable x_i with a large domain, it is possible for its matching value ($M[i]$) to be removed from D_i while $x_i \notin$triggeringVars, thus invalidating

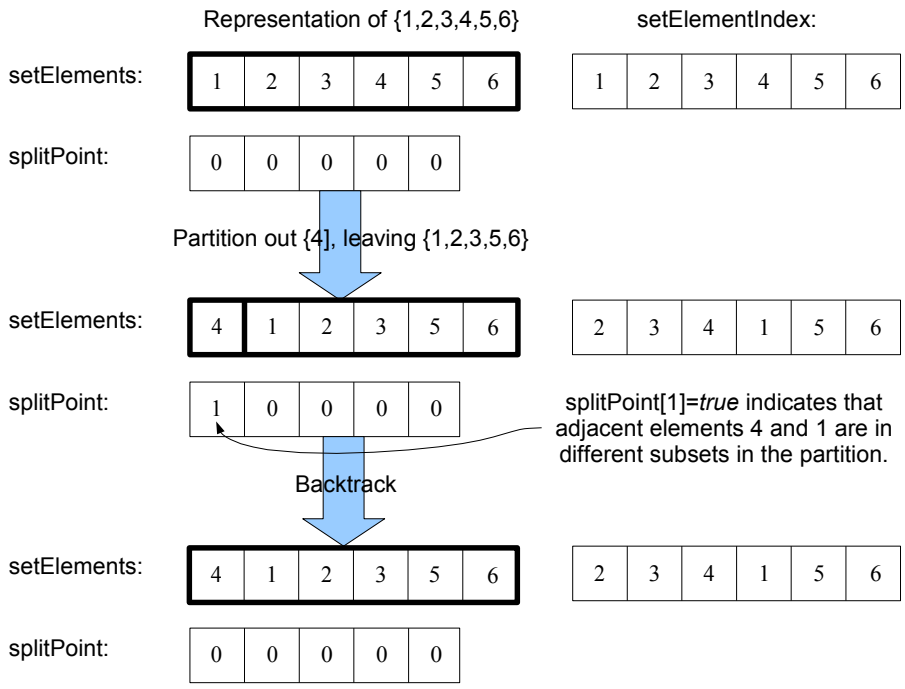


FIGURE 4.3. Illustration of partitioning element 4 from the set $\{1,2,3,4,5,6\}$

the matching without triggering the constraint. To cover this case, lines 15-17 check the matching for an SCC s. If some value in the matching has been removed, FindMaximumMatching is called. This is only required when domain counting is used, so we introduce the flag DomainCounting, which is True iff domain counting is being used.

Propagate-SCC iterates through the set of triggering variables, finding which SCC each variable belongs to (named s, line 3) and checking if the matching has been invalidated (line 4). If it has, FindMaximumMatching is called to repair the matching (lines 5-6).

Lines 7-11 of algorithm 3 implement the assignment optimization. If x_i has been assigned, lines 8-11 are executed. It is possible that s has already been added to changedSCCs, and since it is about to be partitioned it must be removed from changedSCCs. This is done on line 8. s is partitioned into the singleton SCC $\{x_i\}$, and the remainder of s: $s \setminus \{x_i\}$ (line 9). Line 10 performs the removal of the assigned value from all variables in s_2, and s_2 is queued on changedSCCs if necessary (line 11). If x_i is not assigned, and it is possible for s to subdivide ($|s| > 1$) then s is added to changedSCCs (line 13).

If the assignment optimization is not required, lines 7-13 are replaced with a single line which inserts s into changedSCCs.

5. DYNAMIC TRIGGERS FOR THE ALLDIFFERENT CONSTRAINT

By default the AllDifferent constraint would be triggered by any change to any variable domain. However it is possible to identify cases where the SCCs will remain strongly connected, and therefore no pruning can be done. For example, domain counting, as described

Algorithm 3 AllDifferent with SCCs processed independently

```
                propagate-SCC(triggeringVars): returns Boolean
(1)  changedSCCs← ∅
(2)  for x_i ∈triggeringVars:
(3)      s ←findSCC(x_i,SCCs) {find SCC including x_i}
(4)      if not inDomain(x_i, matching[i]):
(5)          hasMatching←FindMaximumMatching(matching, s) {repair the match-
     ing}
(6)          if not hasMatching: return False
(7)      if isAssigned(x_i):
(8)          changedSCCs←changedSCCs\{s}
(9)          Partition s into s_1 = {x_i} and s_2 = s \ {x_i} in SCCs
(10)         for x_j ∈ s_2: removeFromDomain(x_j,getMin(x_i))
(11)         if |s_2| > 1: changedSCCs←changedSCCs∪{s_2}
(12)     else:
(13)         if |s| > 1: changedSCCs←changedSCCs∪{s}
(14) for s ∈changedSCCs:
(15)     if DomainCounting and (∃x_i ∈ s : not inDomain(x_i, matching[i])):
(16)         hasMatching←FindMaximumMatching(matching, s)
(17)         if not hasMatching: return False
(18)     FindSCCsRemoveValues(matching, s)
(19) return True
```

in Section 2.3.2, yields one such approach. As described in Section 2.3.6, Katriel took a different approach, defining important edges in the flow graph as those whose deletion causes further deletions, and showing that the number of important edges is small [16]. Katriel did not, however, extend this observation to a method which can enforce GAC correctly while (possibly) reducing work. We describe in this section a cheap method for finding a set of edges which includes all important edges, and how this can be implemented in a constraint solver using dynamic (movable) triggers.

5.1. **Background.** Gent *et al*. proposed *watched literals* [10], inspired by SAT. Used as triggers to fire constraint propagations, watched literals have three features different from triggers as normally used. Watched literals only cause propagation when a given variable-value pair is deleted; their triggering conditions can be changed dynamically during search; and they remain stable on backtracking so do not use memory for restoration. Watched literals have been shown to be effective for the element, table and Boolean sum constraints by Gent *et al*. [10, 11].

Watched literal propagation algorithms typically revolve around the concept of *support*. A support for a literal is an object which is evidence that the literal is consistent, and therefore cannot be removed by the propagation algorithm. An example of support would be a valid, acceptable tuple of a table constraint.[5] In this case, the tuple can act as support for all the literals it contains. While this support is intact, no work needs to be done, but the constraint must be triggered when any part of the support is invalidated. A second example of support is a pair of unassigned variables for a CNF clause in SAT: this shows that no unit propagation can be done. Therefore it is a support for all literals in the clause.

[5]Each element of a valid tuple is in the relevant domain, and an acceptable tuple is one which satisfies the constraint, as defined in Section 2.1.

The variables have certain *events* which occur when their domain is changed. These could include lowering the upper bound, raising the lower bound, assigning the variable to a single value, or removing a specific value from the domain. We refer to *placing* a watched literal, meaning to attach it to a variable event, and *clearing* it, which removes it from its event. When a variable event occurs, the solver iterates through the watched literals attached to the event, calling the relevant constraint propagator for each. A fixed number of watched literals is allocated by the constraint before search begins.

The concept of watched literals is not completely suitable to the AllDifferent constraint, because watched literals must be correct even though they are not moved on backtracking, a property called "backtrack-stability". The object we use as support is not always backtrack-stable (as we will demonstrate in the next section), therefore the triggers must be restored on backtracking. Gent *et al.* refer to backtracked watched literals as *dynamic triggers* [10].

5.2. **Adapting AllDifferent for dynamic triggers.** When adapting the AllDifferent algorithm to use dynamic triggers, the important structure is the set of SCCs: if each SCC remains strongly connected, then no propagation can be done, and it is not necessary to trigger the constraint. Therefore we focus on Tarjan's algorithm, and identify edges in the residual graph which must be present for Tarjan's algorithm to follow the same trajectory, and therefore return the same result, if it were to be executed again. All important edges, in Katriel's sense [16], must be in this set. Katriel does not give an explicit algorithm for finding a set containing all important edges, although one could be extracted from her proof. The construction of her proof is based on depth first search in each SCC. We follow a very similar approach but present it in the context of Tarjan's algorithm. This method is simple and efficient, only requiring that some information is gathered as Tarjan's algorithm runs.

We collect a set T of edges as follows. As described in Section 3.2, Tarjan's algorithm performs a depth-first search (DFS) in the residual graph $R = \langle V', E' \rangle$. The edges in E' which are traversed by the DFS are included in T. The lowlink value of each vertex is also updated using edges in the graph, and the criterion for identifying an SCC is based on the lowlink value. For each vertex, the lowlink value may be changed several times, but only its final value is used in identifying SCCs, therefore the edge used to obtain its final value is included in T. All other edges in E' are not included in T.

We claim that the removal of any edge in E' and not in T does not affect the set of SCCs. We consider *all* such edges together, since the effect of removing edges on the set of SCCs is monotonic. (Removing an edge may cause no change, or cause an SCC to subdivide into two SCCs. Therefore the number of SCCs monotonically increases.) To demonstrate the claim, we prove that digraph $\hat{R} = \langle V', T \rangle$ has the same SCCs as R. The execution of Tarjan's algorithm on digraph D is denoted $\mathscr{T}(D)$.

Theorem 5.1. *The digraphs $R = \langle V', E' \rangle$ and $\hat{R} = \langle V', T \rangle$ have the same SCCs*

Proof. The proof is by showing that Tarjan's algorithm returns the same SCCs for both R and $\hat{R}$. Tarjan's algorithm is correct [27] (Thm. 14).

The order of vertex exploration in the DFS of Tarjan's algorithm is irrelevant to the result. It is possible for $\mathscr{T}(\hat{R})$ to perform the exact same DFS as $\mathscr{T}(R)$, because the set of vertices is the same and all edges required by the DFS are in T by definition. Hence, without loss of generality, we assume that $\mathscr{T}(\hat{R})$ *does* perform the same DFS as $\mathscr{T}(R)$. Therefore DFSNum for each vertex is identical.

At each vertex v in the DFS tree, lowLink[v] is computed by taking the minimum of a set S_v of values. S_v corresponds to the lowLink or DFSNum of neighbours of v. For each neighbour, the inclusion of its corresponding value in S_v depends only on DFS order, which

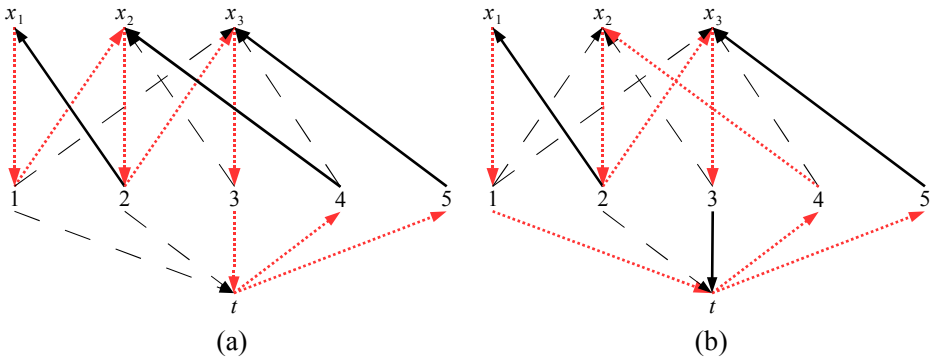


FIGURE 5.1. Example of identifying triggers with two different DFS orderings

is invariant. Therefore, all that is required is that an edge corresponding to a smallest value in S_v is present in T. This is the case by definition.

The final part of $\mathscr{T}(R)$ constructs the SCCs as recursion unwinds. The computed SCCs depend on final values of lowLink and the order of DFS. Therefore $\mathscr{T}(\hat{R})$ constructs the same set of SCCs. □

Some edges in T directly represent a literal: the edge is of the form $x_i \mapsto j$ or $j \mapsto x_i$. We place dynamic triggers on all corresponding literals. It is safe to ignore all other edges in T since they do not correspond to a domain value.

Figure 5.1(a) illustrates the process on a residual graph representing three variables, x_1, x_2, x_3 and five values $1, 2, 3, 4, 5$. The DFS is performed in the following order: $x_1, 1, x_2, 2, x_3, 3, t, 4, 5$. The eight edges traversed by the DFS are represented in wide dotted lines in the figure. Three edges are used to finally change a lowlink value: $2 \mapsto x_1$, $4 \mapsto x_2$, and $5 \mapsto x_3$, which are represented in solid black in the figure.

Edges to and from t are ignored for the purpose of placing triggers. For figure 5.1(a), the set of triggers would be $x_1 \mapsto 1$, $x_1 \mapsto 2$, $x_2 \mapsto 1$, $x_2 \mapsto 2$, $x_2 \mapsto 4$, $x_3 \mapsto 2$, $x_3 \mapsto 3$ and $x_3 \mapsto 5$.

Figure 5.1(b) shows a different execution of the algorithm over the same graph. In this case, DFS is performed in the order $x_1, 1, t, 4, x_2, 2, x_3, 3, 5$. This gives a smaller set of triggers, since $x_2 \mapsto 1$ is not included in this case. All other triggers are the same as for figure 5.1(a).

This method yields at most 3 triggers per variable, plus one trigger per spare value (at most $3r + (d - r) = 2r + d$ triggers), but not all triggers we find may be necessary. For example, in figure 5.1(a), where the triggers on $4 \mapsto x_2$ and $5 \mapsto x_3$ are not necessary. The other triggers are sufficient to prove that the three vertices x_1, x_2, x_3 are indeed in the same SCC.

The triggers are not always backtrack-stable. Consider the case where an SCC s_1 divides into s_2, s_3 when some edges are lost: the triggers computed for both s_2 and s_3 clearly do not cover all the edges necessary to prove the connectedness of s_1, since there are no edges connecting the two components. Therefore, when s_1 is restored upon backtracking, the triggers must be backtracked as well.

We conjecture that this approach will work well when domains are large, or when few values are removed from the domains at each search node.

5.3. **Implementing the collection of dynamic triggers.** This approach is very cheap to implement since it only requires some information to be collected from the execution of Tarjan's algorithm. No additional computation is required.

Two sets of changes are needed to algorithm 2. Firstly, at the top of algorithm FindSCCsRemoveValues, all dynamic triggers for variables in s are cleared. Then a dynamic trigger is placed on each value used in the matching for the current SCC s. This is because these values will be used during the DFS. In addition, a watched literal is placed on each value of the matching. This is required because the matching is not backtracked, and the dynamic triggers are, therefore they can diverge when search backtracks. Placing the watched literals as well guarantees that AllDifferent will be triggered when the matching is violated. In some cases this would cause the constraint to be triggered twice for the same variable-value pair. However, when combining dynamic triggers with a priority queue (Section 2.3.3), when the constraint is triggered it merely adds the triggering variable to a set to be processed later, and returns. Therefore the cost of the additional watched literals is minimal.

The second set of changes are in algorithm TarjanRemoveValues. When curnode$\in \{1 \dots d\}$, dynamic triggers are added corresponding to edges used for the DFS (line 11) and for the final change of lowLink[curnode] (lines 9 and 12). If the final change of lowLink[curnode] occurred on line 12, one edge was used for both the DFS and updating lowLink. It is not necessary to place two dynamic triggers corresponding to one edge. Only one dynamic trigger is placed in this case.

5.4. **Internal dynamic triggers.** It is possible to simulate dynamic triggers entirely within the AllDifferent constraint. To do this, the propagator stores the dynamic trigger values in a backtracking array. The constraint sets static triggers to trigger on any domain change. When it is triggered by a variable x_i, the propagator checks the domain of the triggering variable to see if any important values have been lost. If not, it immediately returns. For each variable, the values are stored contiguously in an array with a length counter. In our experiments, the arrays are backtracked by block copying. We refer to this method as *internal dynamic triggers*.

There are two reasons it might be useful to simulate dynamic triggers in this way. Firstly, it may be more efficient. The cost of writing the trigger value into an array is very low, and clearing an array (by setting the size to 0) is very cheap. By contrast, placing a "real" dynamic trigger (implemented with doubly-linked lists, as described by Gent et al. [10]) requires four assignments to pointers. Clearing a set of dynamic triggers requires two assignments for each literal. In addition to this, any changes must be recorded on the trail stack, and reversed on backtracking. Secondly, a solver may not provide dynamic triggers. Indeed, most solvers do not provide this facility. Therefore internal dynamic triggers are important for the general applicability of dynamic triggers for AllDifferent.

6. EXPERIMENTAL EVALUATION

In this section we describe the context of our experimental evaluation. Then we present four groups of experiments. First we evaluate the standard approaches of incremental matching and priority queueing in Section 6.3. Secondly, in Section 6.4 we evaluate the dynamic triggers and domain counting approaches, which are both intended to reduce the number of calls to the propagator. Thirdly, in Section 6.5 we evaluate processing SCCs independently. Finally we compare our best GAC AllDifferent constraint against a propagator that establishes a weaker consistency.

6.1. **Experimental context.** For all experiments we use the solver Minion [9, 10]. We adapted Minion 0.4.1 in the following ways:

Monotonic set: We made a monotonic set available to constraints through a new interface. By monotonic we mean that once an element of a set is removed at a given node, the element remains out of the set at all descendant nodes. Monotonicity allows an efficient implementation through trailing. We represent each element by the value true or false: since all values move only from true to false, we only need to store the element being changed and not a value to restore it to. This is used to implement the set partition data structure described in Section 4.1. In particular it is used for the splitPoint array. The cost of removing an element from the set combined with its subsequent restoration on backtracking is $O(1)$.

Dynamic triggers: We took watched literals [10] and added a trailing mechanism to restore them as search backtracks. This facility co-exists with standard watched literals. In Minion, a watched literal may be placed on a variable-value pair, an upper or lower bound, or a variable (triggering on any domain change, or triggering only on assignment). A watched literal can also be unused. AllDifferent makes use of watched literals and dynamic triggers on variable-value pairs.

Minion already provides the other facilities that we need. When a constraint is triggered, it is easy to identify the triggering variable. The inDomain, getMin and getMax methods allow us to query the variable domains. The method removeFromDomain is used to remove values from variable domains.

Domain lookups are designed to be fast in Minion, since it is a very common operation. This is important when running the graph algorithms, since they query domains in order to discover the graph as they traverse it. Domain iteration is also important for the AllDifferent algorithm. Unfortunately Minion does not support domain iteration, but it does maintain upper and lower bounds that are used to bound the iteration.

All experiments were run on Apple iMac computers with 2GHz Core Duo processors and 2GB RAM, under OS X Tiger (10.4.11). The branch of Minion which we used is available at `http://minion.sourceforge.net/files/`, including build instructions.

We have conducted extensive testing, which our code has passed. As well as continual and detailed testing during development, we checked that the number of search nodes explored is the same for each variant of the algorithm for each instance, excepting only instances exceeding time or node limits.

In all experiments comparing variants of GAC AllDifferent, we limited search nodes to 500,000, and time to 1,200s. Because of the possibility of timeout, our main metric of speed is nodes searched per second, instead of raw runtime. For some Social Golfers and Golomb instances Minion did not stop until well after the time limit, because of a minor flaw in the time limit implementation.

Finally, all runtimes we report are total time and nodes used to solve or timeout on each given instance, including all initialisation and search time including time outside the AllDifferent propagator. This automatically means that all incidental features of each optimization are accounted for, such as for example additional or reduced memory usage and its effect on practical runtime. It does however mean that results we report are typically less dramatic than would be obtained if we had only measured runtime inside the AllDifferent propagator. Despite this we will often see orders of magnitude improvement in runtime.

6.1.1. *Minion queue mechanisms and triggers.* Minion is a variable-centric solver with an additional constraint-centric queue. Conceptually there are two queues, which are described below. The solver has two queues for efficiency reasons: the variable queue is very fast, because adding a variable event to the queue is an $O(1)$ operation. However, the variable queue does not allow constraints to be given a low priority. Having the additional constraint queue overcomes this limitation.

The variable queue. The variable queue contains variable events, which are of three different types:

- Value a removed from D_i
- Upper/lower bound of x_i changed
- x_i set to a

The variable queue can contain duplicates of the bound events. The other events cannot be duplicated in the queue simply because they cannot happen twice at a single search node. Each variable event has two lists associated with it: a list of static triggers that is fixed before search begins, and a doubly-linked list of dynamic triggers. To propagate a variable event, the solver iterates through both lists, calling the propagators associated with the triggers. Each trigger contains an integer which is passed to the propagator. In this way, the propagator can identify which trigger (and therefore which variable event) has triggered it.

The constraint queue. The constraint queue contains pointers to constraints. Constraints are responsible for setting triggers, and for adding themselves to the constraint queue as necessary. In this way, when a constraint is triggered by the variable queue, it may perform a test to determine whether a record should be added to the constraint queue. This allows us to implement domain counting, described in Section 2.3.2, and internal dynamic triggers as described in Section 5.4.

Indeed the constraint can fail, update internal data structures and perform propagation when triggered from the variable queue, so this mechanism is more general than Lagerkvist and Schulte's advisors (Section 2.3.7) which are not permitted to perform propagation.

The constraint queue allows duplication, however the AllDifferent constraint keeps a record of whether it is present on the constraint queue, and thus avoids duplication.

The constraint queue is not a priority queue. However, the constraint queue has a lower priority than the variable queue: the variable queue is emptied before each item is processed from the constraint queue. In all the experiments presented below, the only constraint to use the constraint queue is the AllDifferent constraint, hence it has a lower priority than any other constraint.

The overall algorithm to process the queues is shown below.

```
while any queue not empty:
    if variable queue not empty:
        process entire variable queue
    if constraint queue not empty:
        process one item from constraint queue
```

6.1.2. *Pairwise AllDifferent.* Minion provides a simple AllDifferent propagator which is triggered whenever a variable becomes assigned, and removes the assigned value from the domains of all other variables. This is clearly a very simple and fast algorithm. It performs the same propagation as AC on a clique of binary not-equal constraints and so does not achieve GAC. We call this the Pairwise propagator.

6.1.3. *Staged AllDifferent.* As described in Section 2.3.4, Schulte and Stuckey propose to combine a cheap propagator (which is similar to the pairwise propagator described above) with GAC AllDifferent [22]. They suggest two ways to do this: simply posting both constraints, or building a staged propagator from the two propagators. They showed staged propagation to be more efficient in Gecode on their instances [22].

The proposed staged propagator makes use of the multiple constraint queues of Gecode, and is impossible to implement exactly in Minion. We implement a close analogue. Whenever the staged propagator is called from the variable queue, it checks if the triggering variable is assigned. If so, the assigned value is removed from the domain of all other variables. Apart from this additional check, the staged propagator is identical to the conventional one. This very simple change yields very good experimental results, shown in Section 6.3.

6.2. **Benchmark set.** We generated a large number of benchmark instances, which are available at `http://minion.sourceforge.net/`. They are described briefly here.

Langford's number problem (prob024 in CSPLib [13]) with $k = 2$ (i.e. two occurrences of each number) and $n \in \{10, 11, 12, \ldots, 25\}$. This is modelled with a vector v of length $2n$ which is all different, where elements $v[i]$ and $v[i+n]$ represent the two positions of colour i in the problem. The model is by Gent, Miguel and Rendl [12]. The variable order follows the indices of vector v, and the value order is ascending.

The Golomb ruler problem (prob006 in CSPLib) is to construct a set of n integers which are all different, and the intervals between pairs are all different. The lowest integer is assumed to be 0, and the highest integer is minimized using branch and bound. This is modelled as a vector of $n(n-1)/2$ differences between pairs of integers, with a single AllDifferent constraint on the vector. The variable order follows the indices of the vector, and the value order is ascending.

Balanced quasigroup with holes (QWH) [17] is the problem of completing a partial latin square with a particular structure. The instances were generated from random, complete latin squares of order $n \in \{20, 25, 30, 35\}$. Ten latin squares were generated at each size, and $\lceil 1.7 \times n^{1.55} \rceil$ holes were punched to create balanced partial latin squares. The number of holes yields instances at or near the difficulty peak. The problem is modelled as an $n \times n$ matrix of variables with domain $\{1 \ldots n\}$, with an AllDifferent constraint on each row and column. The variable order is left to right along the rows. The rows are searched in sequence down the matrix. The value order is ascending.

Quasigroup existence (prob003 in CSPLib) is the problem of determining whether a quasigroup exists with certain properties, for example idempotence ($a \times a = a$ for all elements a). We used types QG3 and QG4, both idempotent and non-idempotent, with orders $n \in \{7, 8, 9, 10\}$ making 16 instances in total. The problem is modelled with an $n \times n$ matrix of variables with domain $\{1 \ldots n\}$, with an AllDifferent constraint on each row and column, an AllDifferent on the primary diagonal, and various other constraints representing the properties of the quasigroup type. Various implied constraints are also included. The model is by Colton and Miguel [5]. The variable and value ordering is the same as for QWH.

Social golfers (prob010 in CSPLib) is the problem of assigning gs golfers to s sets of size g, for each of w weeks, such that two golfers never play together more than once. It is modelled as w vectors of variables with domain $1 \ldots g \times s$, representing the weeks. For each week, the vector is partitioned into s sets. To break some symmetries, the sets are lex-ordered within the week, golfers are ordered within the sets and the weeks are lex-ordered. Each week vector has an AllDifferent constraint. A second vector of variables with domain

$\{1 \dots gs(gs-1)/2\}$ represents the pairs that play together. For each week, for each pair of variables in the same set, the two variables are mapped to a single variable in the pairs vector using a table constraint. The pairs vector has an AllDifferent constraint on it. We generated social golfers instances with $g = 4$ and the following other parameters $\langle w,s \rangle$: $\langle 5,4 \dots 8 \rangle \langle 6,4 \dots 8 \rangle$, $\langle 7,5 \dots 8 \rangle$, $\langle 8,6 \dots 8 \rangle$, $\langle 9,7 \dots 8 \rangle$, $\langle 10,8 \rangle$, making 20 instances in all. The variable order is to search each week in turn, and for each week we follow the indices of the w vector. The value order is ascending.

Sports scheduling is similar to social golfers where $g = 2$ (i.e. fixtures involve two teams). n teams play on $n/2$ pitches over $n-1$ weeks. Each team plays on each pitch at most twice. It is modelled as $n-1$ vectors of variables with domain $1 \dots n$ representing the weeks. The vector is partitioned into pairs. To break some symmetries, each pair is ordered and the weeks are lex-ordered. Also, the pitches are interchangeable so the vectors of games on each pitch are lex-ordered. Sports scheduling also has the pairs vector with AllDifferent, and the table constraints between the two representations. In this way it is guaranteed that every team plays every other team exactly once. The variable and value order is the same as for social golfers.

The contrived problem has been contrived to show the benefit of dynamic triggers and domain counting. It is pathological because the GAC AllDifferent constraint performs no pruning, despite doing a significant amount of computation. It consists of two vectors v and w. v is of length 5 and the variables have domain $\{1 \dots 50\}$. An AllDifferent constraint (using the pairwise propagator) is placed on v, and also $v[4] = v[5]$. Therefore there are no solutions. The pairwise AllDifferent is only present to make the problem unsatisfiable, while causing Minion to search extensively: the binary search tree has $50 \times 49 \times 48 \times 47 = 5527200$ left branches. w is a vector of length $l \geq 4$, containing variables with domain $\{1 \dots d\}$. A GAC AllDifferent constraint is placed on w, and the two vectors v,w are linked by $v[1] \neq w[1]$, $v[2] \neq w[2]$, $v[3] \neq w[3]$, and $v[4] \neq w[4]$. Hence, whenever a variable in v is assigned, one value is removed from a variable in w by propagation. The variable order follows the indices of v, and the value order is ascending.

We generated instances with $l = \{100, 200, 300, 400, 500\}$, of two types where $d = l$ or $d = l+1$. Both types should work well with dynamic triggers, because only one value is removed for each left branch, and this is unlikely to trigger the GAC AllDifferent. When using domain counting, when $d = l$ the constraint is always triggered by the removal of a single value. However when $d = l+1$ then the constraint is never triggered, so the propagator is only executed at the root node. In this case, domain counting should outperform dynamic triggers.

6.3. **Experiment one: variants proposed in the literature.** Prioritized queueing and incremental matching are standard techniques. In this experiment we test their merit. We also consider an alternative matching algorithm, and staged propagation.

Simple: The simplest variant of AllDifferent is shown in algorithm 1. It does not use the constraint queue, therefore it is called once for each variable event. Simple does not process SCCs independently or use dynamic triggers or domain counting. The Hopcroft-Karp algorithm is used to compute the matching.

PriorityQ: The Simple algorithm, but called from the constraint queue. It is added to the constraint queue on any variable event, unless it is already present. Therefore duplicates are removed and the constraint is propagated after all others.

PriorityQ-IncMatch: PriorityQ with incremental matching. (Algorithm 1 with line 1 removed, so that the matching is retained from one call to the next.)

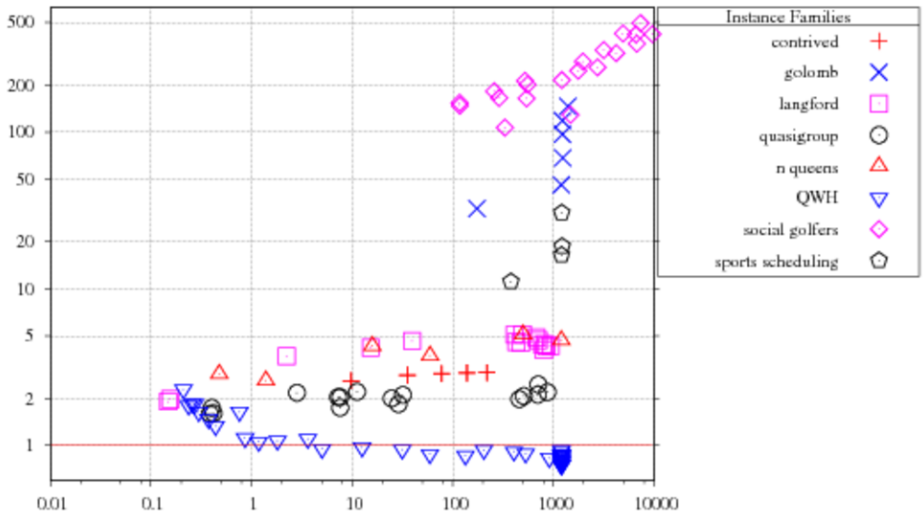


FIGURE 6.1. Speedup of PriorityQ over Simple. The graph is a scatter-plot, with each point comparing results on a single instance. The x-axis represents the run time of Simple to solve the instance. The y-axis gives the speedup obtained by using PriorityQ instead of Simple. A ratio of 1 indicates that the two methods run at the same speed, with ratios higher than 1 indicating that PriorityQ is faster, and ratios less than 1 indicating that Simple is faster. The ratio is calculated by dividing the number of search nodes explored per second by PriorityQ by that for Simple. In this graph we can see that performance ranges from a slight slowdown from using PriorityQ, on some QWH instances, to speedups of hundreds of times on some social golfers instances.
All subsequent graphs labelled 'Speedup of X over Y' follow the same conventions, where in this case X=PriorityQ and Y=Simple.

PriorityQ-IncMatch-BFS: This is PriorityQ-IncMatch using the FF-BFS matching algorithm rather than Hopcroft-Karp.

PriorityQ-IncMatch-BFS-Staged: This is PriorityQ-IncMatch-BFS with staged propagation as described in Section 6.1.3.

Firstly, Simple and PriorityQ were compared on our benchmark set. We expected PriorityQ to perform better for all instances. Figure 6.1 shows that this is not the case, although most instances benefit from the constraint queue, with some performing over 100 times better. This mainly agrees with the results of Schulte and Stuckey [22] regarding priority queueing, although their results are less dramatic.

The instances that Simple solves faster than PriorityQ are all QWH instances. These contain only AllDifferent constraints, so reducing the priority of the constraint would have no effect.

Given the theoretical advantage of incremental matching, we expect PriorityQ-IncMatch to perform better than PriorityQ. This is the case for all instances, as shown in figure 6.2, although the gain is less than 40%.

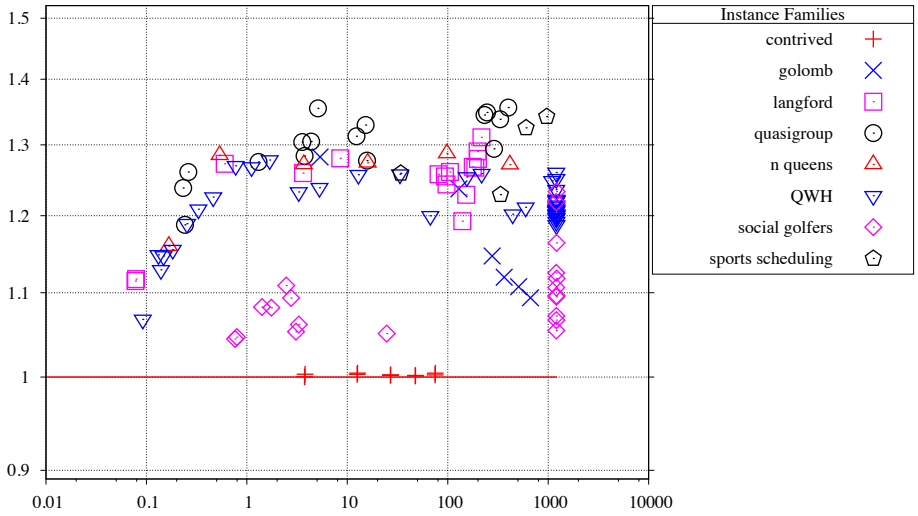


FIGURE 6.2. Speedup of PriorityQ-IncMatch over PriorityQ

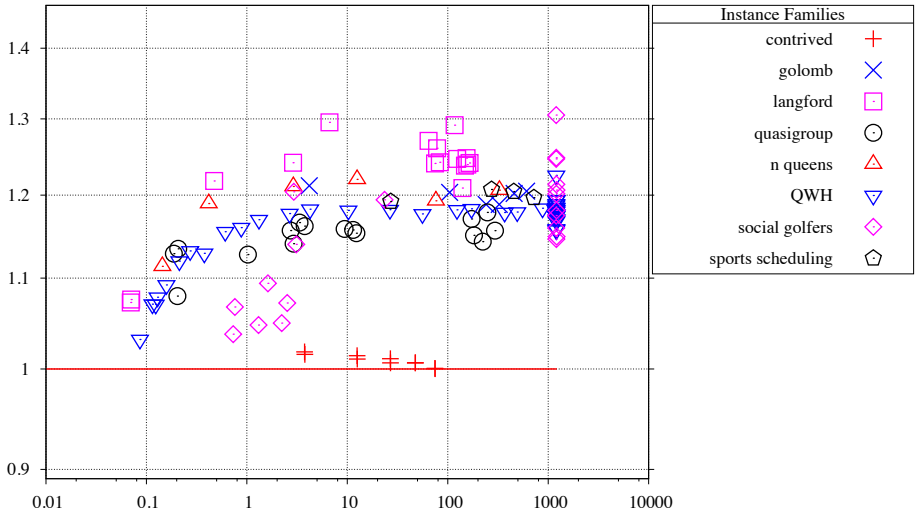


FIGURE 6.3. Speedup of PriorityQ-IncMatch-BFS over PriorityQ-IncMatch

We compared PriorityQ-IncMatch-BFS with PriorityQ-IncMatch to compare the two matching algorithms. As shown in figure 6.3, the AllDifferent with FF-BFS can be 30% faster, and is never slower than with Hopcroft-Karp.

Finally, we compare the staged propagator PriorityQ-IncMatch-BFS-Staged to PriorityQ-IncMatch-BFS. As shown in figure 6.4, staged propagation is very useful for our set of benchmarks, with up to three times improvement.

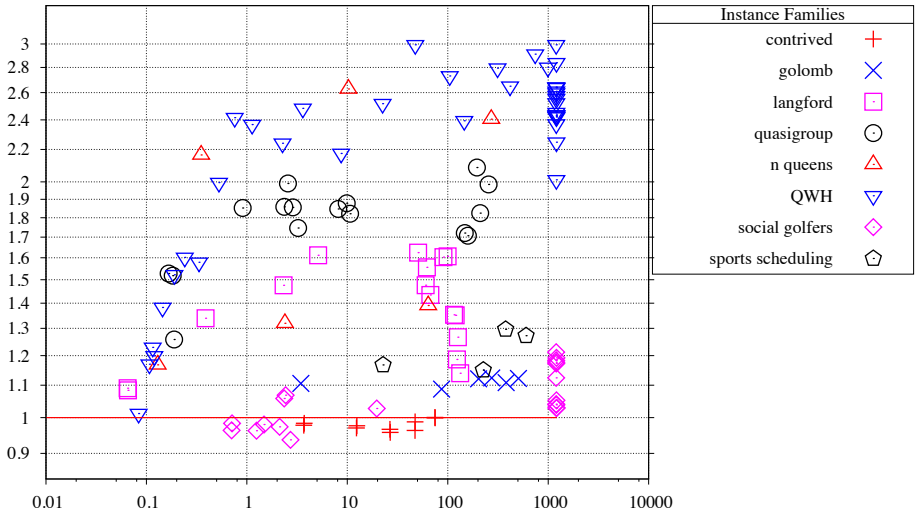


FIGURE 6.4. Speedup of PriorityQ-IncMatch-BFS-Staged over PriorityQ-IncMatch-BFS

For all further experiments, we use both the constraint queue and incremental matching, since they are standard techniques, and verified to be useful in this context. We use the FF-BFS algorithm in all further experiments, since it is often faster and never slower than Hopcroft-Karp. We also use staged propagation since it is considerably faster on average. Comparing PriorityQ-IncMatch-BFS-Staged to Simple, we observe a speedup between 2.14 and 863 times, with a mean average speedup of 80.8, excluding the contrived instance family.

6.4. **Experiment two: Dynamic triggers and domain counting.** For the purpose of this experiment, all variants will use the constraint queue and incremental matching. The aim is to compare waking up on all domain events against using dynamic triggers and domain counting.

Baseline: The same as PriorityQ-IncMatch-BFS-Staged in the previous section.

DynamicTrigger: Baseline with the addition of dynamic triggers as described in Section 5.

DynamicTriggerInternal: Baseline with the addition of internal dynamic triggers (Section 5.4).

DomainCount: When the constraint is triggered from the variable queue, and it is not present on the constraint queue, the domain D_i of the triggering variable is counted. (The number of values is not maintained by default in Minion.) If $|D_i| < r$ then the constraint is added to the constraint queue.

Figure 6.5 shows the ratio of search nodes per second between DynamicTrigger and Baseline. Instances are scattered above and below 1, suggesting that the advantage of DynamicTrigger is negated by its overheads in many cases. All the contrived instances are considerably faster with DynamicTrigger, as expected. Figure 6.6 shows the same plot between DynamicTriggerInternal and Baseline. DynamicTriggerInternal performs slightly better than Baseline on most instances, exploring up to 1.3 times as many nodes per second, with an average 6% improvement (excluding contrived). Surprisingly, the contrived

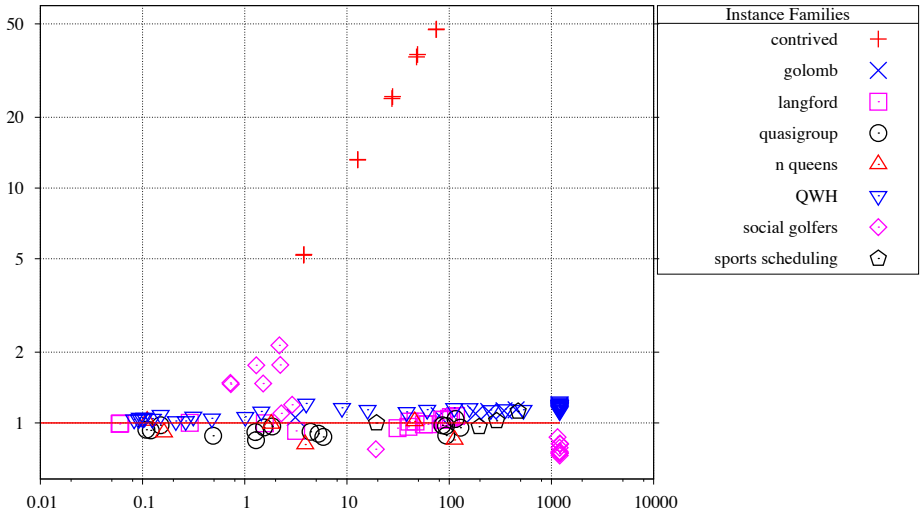


FIGURE 6.5. Speedup of DynamicTrigger over Baseline

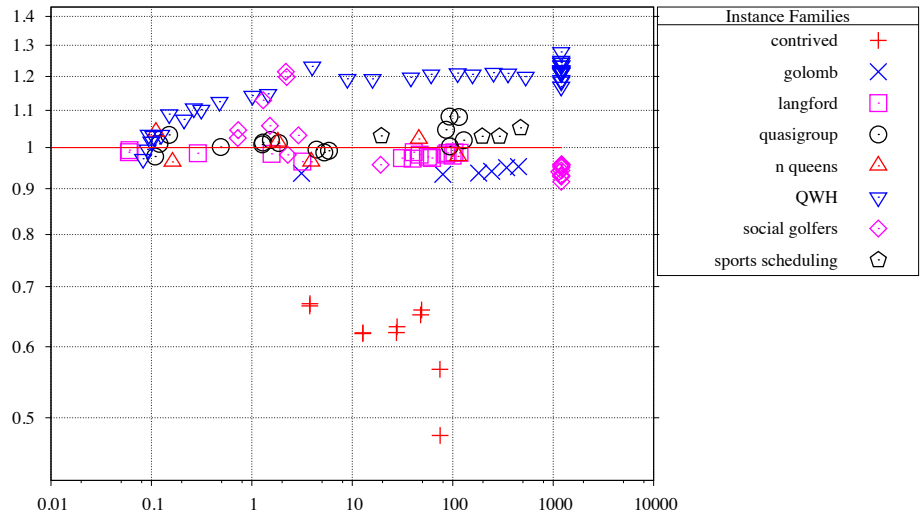


FIGURE 6.6. Speedup of DynamicTriggerInternal over Baseline

instances are an exception, since they are slower with DynamicTriggerInternal than Baseline. In DynamicTriggerInternal, the cost of placing dynamic triggers is much lower (since they are just written into an array). However, the arrays are backtracked by block copying, and for the contrived instances the arrays are large.

Finally we compare DomainCount with Baseline. The ratio of search nodes per second is shown in figure 6.7. Domain counting is cheap, so no instances are substantially slower, but it never substantially wins either. This is perhaps not surprising, since domain counting was originally intended for set or tuple variables with very large domains [20]. The

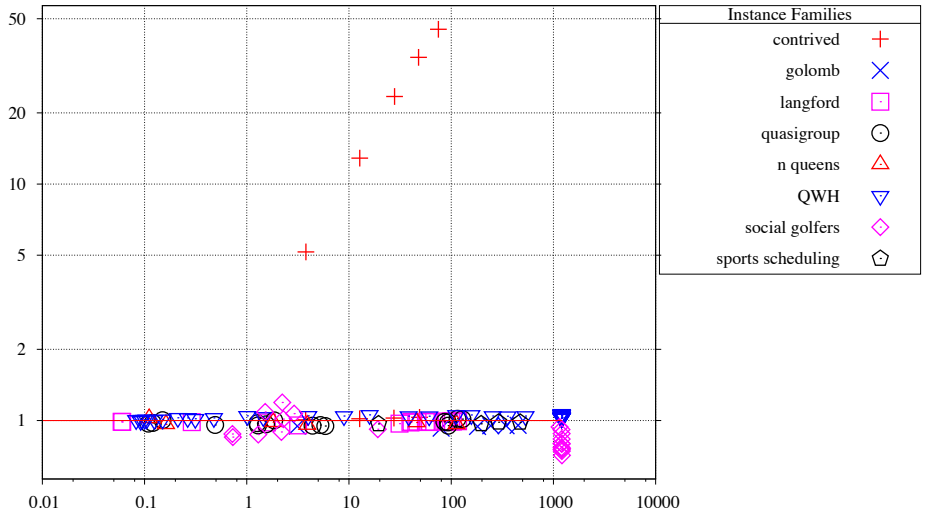


FIGURE 6.7. Speedup of DomainCount over Baseline

contrived instances behave as expected, with instances where $d = r + 1$ showing a huge advantage for domain counting, and instances where $d = r$ showing no advantage.

Our conclusions for this experiment are that internal dynamic triggers are worthwhile on average, whereas dynamic triggers had too great an overhead and domain counting does not work well on this benchmark set.

6.5. **Experiment three: Processing SCCs independently.** The SCC optimization described in Section 4 aims to decrease the time spent running the graph algorithms. It is independent of domain counting. However, there is a dependence between dynamic triggers and the SCC optimization, because running Tarjan's algorithm on a smaller graph will potentially cause fewer triggers to be moved, therefore potentially reducing the overhead of using dynamic triggers. For these experiments we ignore domain counting but do consider dynamic triggers.

Baseline: The same as PriorityQ-IncMatch-BFS-Staged in experiment one.
SCC: In addition to Baseline, SCCs are processed independently as described in Section 4.
SCC-AssignOpt: In addition to SCC, the assignment optimization described in Section 4.2 is used.
SCC-AssignOpt-DynamicTrigger: In addition to SCC-AssignOpt, dynamic triggers are used.
SCC-AssignOpt-DynamicTriggerInternal: In addition to SCC-AssignOpt, internal dynamic triggers are used.

Figure 6.8 shows results comparing SCC to Baseline. The SCC variant is able to explore up to ten times more search nodes per second on the benchmarks, and SCC is never slower than Baseline. This is as expected, since there is not much additional cost with SCC, and the potential savings of running the graph algorithms on smaller graphs are large.

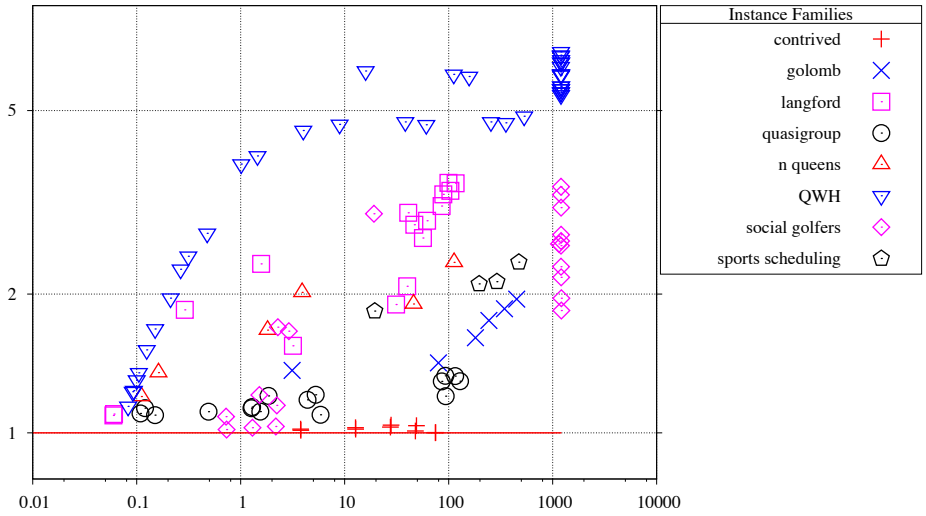


FIGURE 6.8. Speedup of SCC over Baseline

Comparing SCC-AssignOpt to SCC (figure 6.9) shows that AssignOpt is worthwhile more often than not, although the mean improvement in nodes/second is only 3%, excluding contrived instances.

Social golfers and sports scheduling problems solve slower with AssignOpt. These two problems have a similar structure, with a very large AllDifferent constraint on the pairs vector. This may indicate that AssignOpt does not scale well to large constraints.

In a staged constraint, part of the work done in the assignment optimization (removing the assigned value from other variables in the SCC) is redundant and could be removed. This could improve AssignOpt a little.

Figure 6.10 shows results comparing SCC-AssignOpt-DynamicTrigger against SCC-AssignOpt. Apart from a few Social Golfers instances, these results are not promising for dynamic triggers. Many instances solve considerably slower with dynamic triggers. Internal dynamic triggers (figure 6.11) fare better, perhaps because the cost of moving triggers is lower. The sports scheduling instances solve faster with internal dynamic triggers. However on average (excluding the contrived problem) internal dynamic triggers lose by 5%. This is in contrast to the experiment without SCC and AssignOpt, where internal dynamic triggers gained 6%. Exploiting SCCs and the assignment optimization have reduced the cost of applying the graph algorithms, so now it is not worthwhile to apply internal dynamic triggers.

Since dynamic triggers do not do well in this context, we regard SCC-AssignOpt as our strongest variant overall.

6.6. **Experiment four: Comparing with the pairwise propagator.** We compare the pairwise propagator (described in Section 6.1.2) with the most efficient variant of GAC AllDifferent. Since the two propagators do not provide the same level of consistency, comparing node rates would be of limited use: it would only show the overhead of maintaining GAC, without showing the benefit. Therefore we compare solution times, using a much longer 7200 second time limit, and no node limit. The variant of GAC AllDifferent we

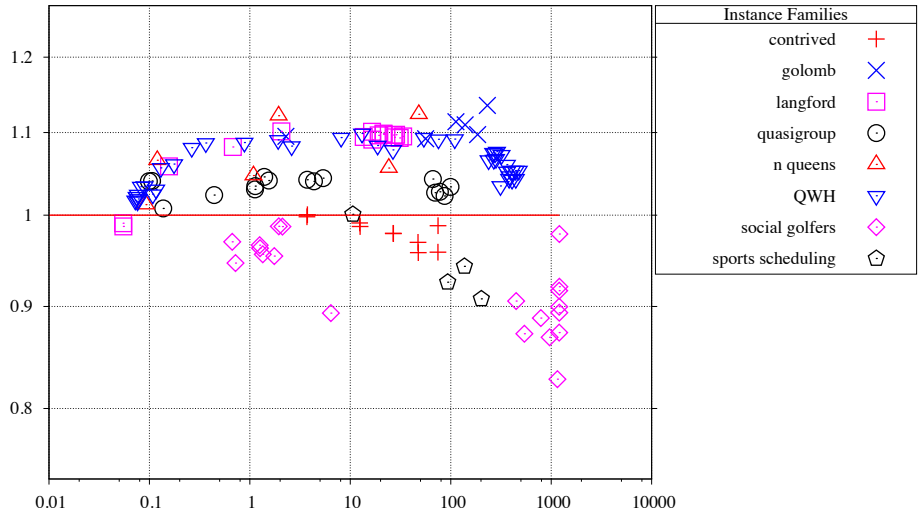


FIGURE 6.9. Speedup of SCC-AssignOpt over SCC

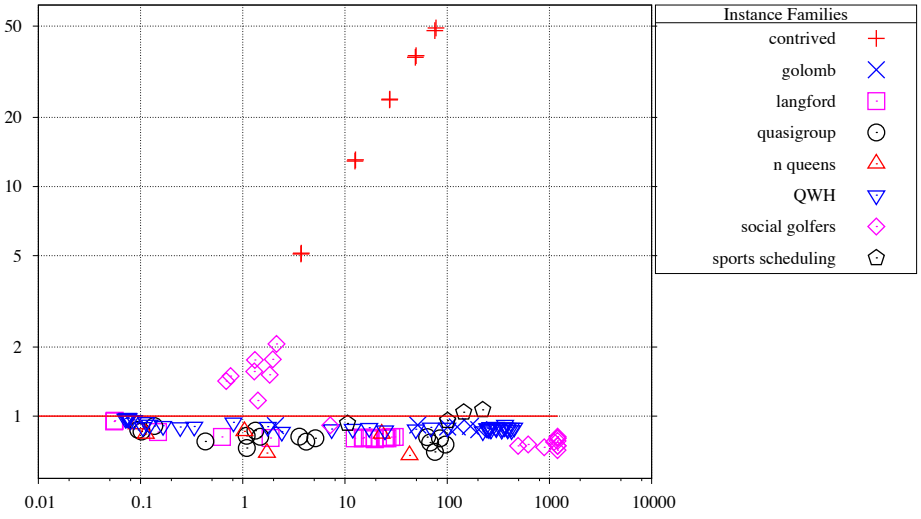


FIGURE 6.10. Speedup of SCC-AssignOpt-DynamicTrigger over SCC-AssignOpt

use is SCC-AssignOpt from experiment three. We refer to this as Best, and to the pairwise propagator as Pairwise.

Figure 6.12 shows a plot of solution times, with Pairwise on the x-axis, and the ratio Pairwise over Best on the y-axis. Points on the line $x = 7200$ represent instances which timed out for Pairwise. Two instances timed out for GAC and not for Pairwise, this is in the area where $x > 1000$ and $y < 1$.

Clearly the GAC reasoning on the AllDifferent constraint is very important for some instances. In particular, QWH and some difficult Social Golfers instances solve much

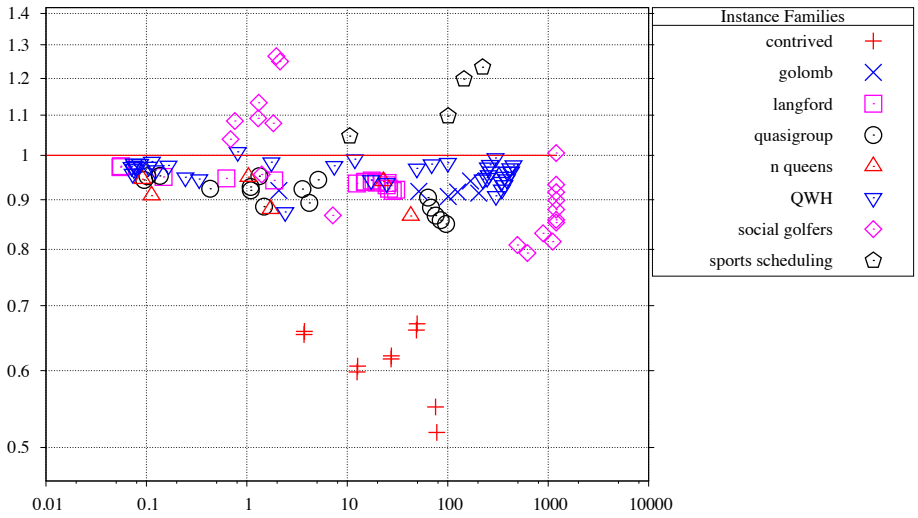


FIGURE 6.11. Speedup of SCC-AssignOpt-DynamicTriggerInternal over SCC-AssignOpt

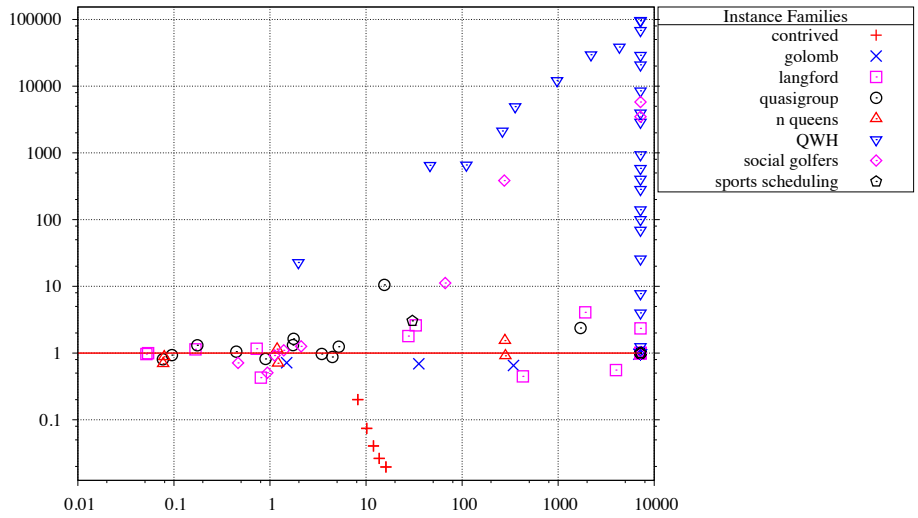


FIGURE 6.12. Comparing Best to Pairwise runtime. In this plot the x-axis represents the runtime of Pairwise, and the y-axis is the ratio Pairwise over Best runtime. The time limit was set to 7200s, and there was no node limit.

faster with GAC. In some cases, they solve more than 1000 times faster than with Pairwise. Many authors (for example Stergiou and Walsh [26]) have found that GAC AllDifferent is important.

There are a large number of instances which caused both GAC and Pairwise to time out. There are two instances where GAC timed out and Pairwise completed (Golomb ruler

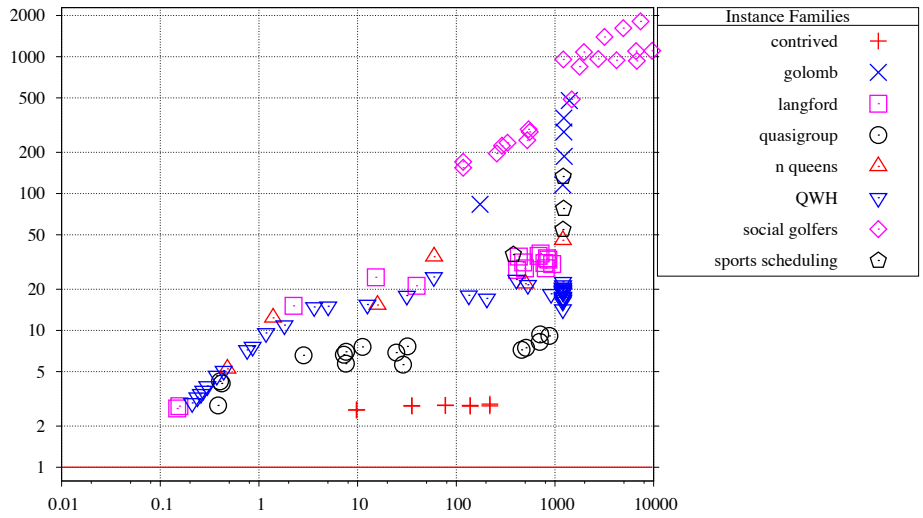


FIGURE 6.13. Speedup of Best over Simple

$n = 13$, Langford's $n = 14$), but there are 22 instances where Pairwise timed out and GAC completed, although many of these are in the QWH class. For many other instances, GAC does not perform as well as Pairwise. However, ignoring the contrived instances, Pairwise is never more than 2.34 times faster than GAC. Interestingly, this holds for easy and difficult instances, without increasing divergence as the instances become more difficult.

Overall this experiment shows that adding the GAC AllDifferent propagator substantially extends the reach of Minion to solve challenging problem instances.

6.7. **Experimental conclusions.** We have individually evaluated many different efficiency measures for the GAC AllDifferent algorithm. In this section, we consider the effect of them all together. Figure 6.13 compares Best with Simple. Including the contrived instances, we get a speedup from 2.69 times to 1813 times, with an average of 168 times. Clearly Best is a huge improvement over Simple, and this shows the importance of taking care to implement AllDifferent well.

Simple does not include standard optimizations (prioritized queueing, incremental matching or staging), and it uses Hopcroft-Karp rather than the more efficient FF-BFS to perform the matching. Therefore in figure 6.14 we compare Best with Baseline. The only differences between these two algorithms are optimizations we have described in Section 4 on exploiting SCCs. All except the pathological contrived instances lie between 0.96 and 7.06 times faster, and the mean improvement is 2.98 times.[6] Since these figures are for solving the instance, the improvement in the AllDifferent constraint is greater than that.

Finally, figure 6.15 is a plot of the nodes explored per second by Best. This is to give an idea of the speed of the algorithm on different classes of instances. The Langford's instances are very fast, exceeding 20,000 nodes per second in some cases, which is perhaps remarkable when maintaining GAC. Social Golfers and QWH are the slowest classes, since

[6]Note that the mean ratios are not multiplicative, since we have Best:Simple = 168, Best:Baseline = 2.98 and Baseline:Simple = 81. Geometric means retain multiplicative properties of the mean. The geometric means are: Best:Simple = 31.98, Best:Baseline = 2.44, Baseline:Simple = 13.11.

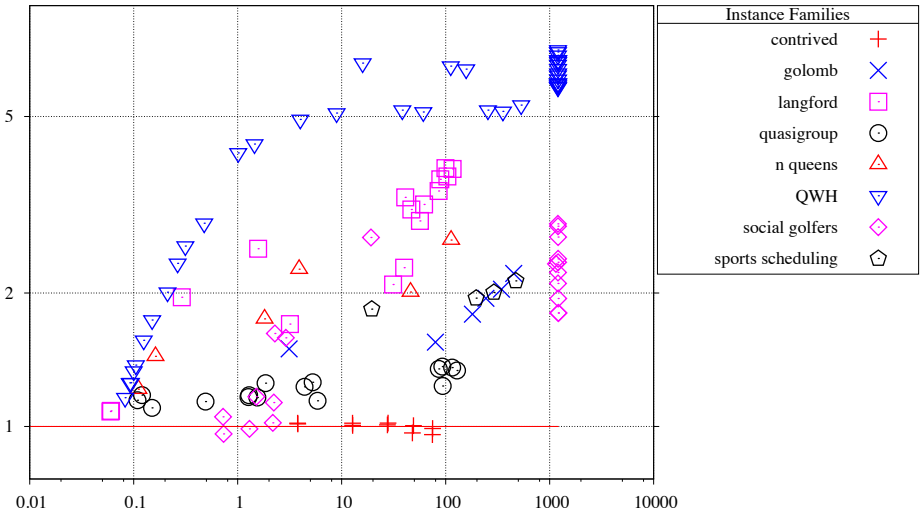


FIGURE 6.14. Speedup of Best over Baseline

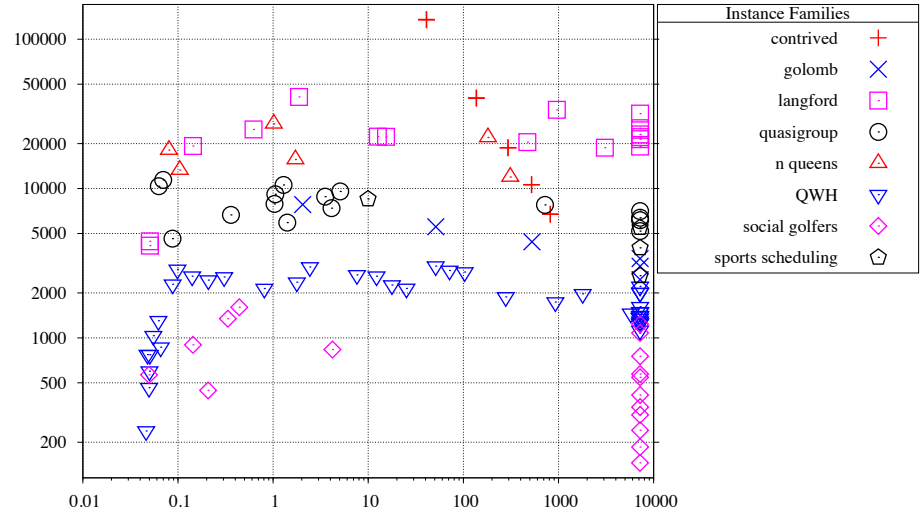


FIGURE 6.15. Plot of Best nodes per second. The x-axis represents Best runtime, and the y-axis the number of nodes searched per second.

Social Golfers has large AllDifferent constraints ($r = 480$, $d = 496$ for the largest instance) and QWH has a large number of them per instance (70 constraints where $r = d = 35$ for the largest instances).

7. Implementation Advice

In this section we abstract from the details of our experimental results to give brief advice to those following us in implementing GAC for AllDifferent, or researching further optimizations for it.

Our results show that there is huge benefit from using the following optimizations, in order of importance: propagating AllDifferent in a separate queue from other constraints; the incremental exploitation of strongly connected components; and combining GAC and pairwise propagators using staging. There is a lesser but still important improvement of 20-30% from incremental matching. Our results on these optimizations are significant enough that they are unlikely to be reversed by different implementation choices or the study of different instances, and these optimizations remain effective when combined. We find that our assignment optimization is worthwhile, although the effect is small enough at 3% it could be omitted to save programmer time without undue penalty for the end-user.

Some of our results depend on the context in which GAC AllDifferent will be used. We did not find dynamic triggers of either type to be worthwhile. In our experiments dynamic triggers were slightly better when implemented internally within the AllDifferent constraint, an approach which is also more portable. We did not find Lagerkvist and Schulte's variant of Quimper and Walsh's domain counting to be beneficial, even with our slight improvement on it. All these conclusions apply only if we ignore our family of contrived instances, on which maintaining GAC adds costs without any benefit. On these instances, both dynamic triggers and domain counting come into their own and internal dynamic triggers were less effective than those built into Minion. Therefore, dynamic triggers and domain counting could be considered if a GAC AllDifferent propagator will be used when it is often ineffective, to ameliorate the overheads in this case.

One question we have not been able to come to a definite conclusion on is the best matching algorithm within the GAC AllDifferent propagator. We experimented with the simpler BFS algorithm and the more complex Hopcroft-Karp algorithm. We found BFS to be significantly better and can recommend it over Hopcroft-Karp. However, there are many other matching algorithms and it is quite possible that another choice might outperform both of the algorithms we tested.

As noted in the introduction, we have not tested against propagators that maintain bounds and range consistency for AllDifferent. Therefore, we cannot advise on the relative merits of those techniques compared to maintaining GAC. We would be delighted to see a study evaluating optimizations of those algorithms along similar lines to this paper, to enable the fairest comparison with our optimized GAC implementation.

8. Conclusions

We have presented an extensive survey of propagation methods to establish generalised arc consistency (GAC) for the AllDifferent constraint using Régin's algorithm. We have surveyed many optimizations proposed in the literature and some methods that have not been previously reported. For most of these methods, we have reported on their implementation and given an empirical analysis of their behaviour. We have paid particular attention to evaluating optimizations in combination with each other, which is (very naturally) not usually a feature of papers which propose optimizations. Our experiments are very easily the deepest experimental analysis of GAC algorithms for AllDifferent. Based on them we have provided advice on which optimizations are key and which are less generally useful.

We would draw particular attention to our results on processing strongly connected components during the search process. We showed significant improvements from the

observation that individual strongly connected components can be treated independently. This paper gives the first detailed report and empirical evaluation of this technique. We also showed a lesser improvement to maintaining strongly connected components by treating the assignment of a variable as a special case, which we believe to be a new optimization. These two techniques speed up search on instances containing AllDifferent by an average of 3 times.

For the best combination of optimizations, we found a mean improvement of more than 160 times in runtime over a careful but unoptimized implementation of Régin's algorithm. Our results confirm that optimizations are not optional extras, but a vital part of a serious implementation of GAC AllDifferent. Our results also show that GAC propagation can be used very widely. Apart from some pathological examples, GAC propagation of AllDifferent never slows down search by more than a factor of 2.34, compared with a highly optimized implementation of the pairwise AllDifferent propagator. The combination of optimizations we have studied and their careful implementation brings GAC propagation of AllDifferent to the point where it is practical for almost all instances, and beneficial for a very large number.

Acknowledgements. We would like to thank Chris Jefferson for technical help with development in Minion [9] and helpful discussion, Andrea Rendl for use of Tailor [12], and Max Neunhöffer for useful discussion of data structures. We thank all our reviewers from *Artificial Intelligence* for their very helpful comments. Our work was supported by EPSRC EP/E030394/1. Ian Miguel is also supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship.

REFERENCES

[1] Abderrahamane Aggoun, David Chan, Pierre Dufresne, Eamon Falvey, Hugh Grant, Warwick Harvey, Alexander Herold, Geoffrey Macartney, Micha Meier, David Miller, Shyam Mudambi, Stefano Novello, Bruno Perez, Emmanuel van Rossum, Joachim Schimpf, Kish Shen, Periklis Andreas Tsahageas, and Dominique Henry de Villeneuve. Eclipse user manual release 5.10, 2006. http://eclipse-clp.org/.

[2] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m/\log n})$. *Inf. Process. Lett.*, 37(4):237–240, 1991.

[3] Claude Berge. *Graphs and Hypergraphs*. North-Holland Publishing Company, 1973.

[4] Mats Carlsson and Christian Schulte. Finite domain constraint programming systems, 2002. Available from ftp://ftp.sics.se/pub/isl/papers/FD%20Systems.pdf.

[5] Simon Colton and Ian Miguel. Constraint generation via automated theory formation. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*, pages 575–579, 2001.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[7] Marie-Christine Costa. Persistency in maximum cardinality bipartite matchings. *Operations Research Letters*, 15(3):143–149, 1994.

[8] David Eppstein. Implementation of Hopcroft-Karp algorithm in Python, 2002. http://www.ics.uci.edu/∼eppstein/.

[9] Ian P. Gent, Chris Jefferson, and Ian Miguel. Minion: A fast, scalable, constraint solver. In *Proceedings 17th European Conference on Artificial Intelligence (ECAI 2006)*, pages 98–102, 2006.

[10] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *Proceedings 12th International Conference on the Principles and Practice of Constraint Programming (CP 2006)*, 2006.

[11] Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the Twenty Second Conference on Artificial Intelligence (AAAI-07)*, pages 191–197, 2007.

[12] Ian P. Gent, Ian Miguel, and Andrea Rendl. Tailoring solver-independent constraint models: A case study with essence' and minion. In *Proceedings of SARA 2007*, pages 184–199, 2007.

[13] Brahim Hnich, Ian Miguel, Ian P. Gent, and Toby Walsh. CSPLib: a problem library for constraints. http://csplib.org/.

[14] J.E. Hopcroft and R.M. Karp. An $n^{2.5}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[15] *ILOG Solver 6.0 User Manual*. ILOG S.A., 2003.

[16] Irit Katriel. Expected-case analysis for delayed filtering. In J. Christopher Beck and Barbara M. Smith, editors, *CPAIOR*, volume 3990 of *Lecture Notes in Computer Science*, pages 119–125. Springer, 2006.

[17] Henry A. Kautz, Yongshao Ruan, Dimitris Achlioptas, Carla P. Gomes, Bart Selman, and Mark E. Stickel. Balance and filtering in structured satisfiable problems. In *Proceedings of IJCAI-2001*, pages 351–358, 2001.

[18] D.E. Knuth and A. Raghunathan. The problem of compatible representatives. *SIAM Journal of Discrete Mathematics*, 5(3):422–427, 1992.

[19] Mikael Z. Lagerkvist and Christian Schulte. Advisors for incremental propagation. In *Proceedings 13th Principles and Practice of Constraint Programming (CP 2007)*, pages 409–422, 2007.

[20] Claude-Guy Quimper and Toby Walsh. The all different and global cardinality constraints on set, multiset and tuple variables. In *Recent Advances in Constraints, LNAI 3419*, 2006.

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

[22] Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *Tenth International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633, Toronto, Canada, September 2004. Springer-Verlag.

[23] Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Artificial Intelligence*, pages 118–132. Springer-Verlag, 2006.

[24] João C. Setubal. New experimental results for bipartite matching. In *Proceedings of netflow93, tech. report TR-21/93, Dipartamento de Informatica, Università di Pisa*, pages 211–216, 1993.

[25] João C. Setubal. Sequential and parallel experimental results with bipartite matching algorithms. Technical Report EC-96-09, Institute of Computing, University of Campinas, Brasil, 1996.

[26] Kostas Stergiou and Toby Walsh. The difference all-difference makes. In *Proceedings IJCAI'99*, 1999.

[27] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[28] Willem-Jan van Hoeve. The alldifferent constraint: A survey. In *Proceedings Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.

[29] Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.

SCHOOL OF COMPUTER SCIENCE, UNIVERSITY OF ST ANDREWS, ST ANDREWS, FIFE KY16 9SX
E-mail address: {ipg,ianm,pn}@cs.st-andrews.ac.uk