The Extended Global Cardinality Constraint: An Empirical Survey

Peter Nightingale[a]

[a]*School of Computer Science*
University of St Andrews
St Andrews
Fife KY16 9SX
United Kingdom

Abstract

The Extended Global Cardinality Constraint (EGCC) is a vital component of constraint solving systems, since it is very widely used to model diverse problems. The literature contains many different versions of this constraint, which trade strength of inference against computational cost. In this paper, I focus on the highest strength of inference usually considered, enforcing generalized arc consistency (GAC) on the target variables. This work is an extensive empirical survey of algorithms and optimizations, considering both GAC on the target variables, and tightening the bounds of the cardinality variables. I evaluate a number of key techniques from the literature, and report important implementation details of those techniques, which have often not been described in published papers. Two new optimizations are proposed for EGCC. One of the novel optimizations (*dynamic partitioning*, generalized from AllDifferent) was found to speed up search by 5.6 times in the best case and 1.56 times on average, while exploring the same search tree. The empirical work represents by far the most extensive set of experiments on variants of algorithms for EGCC. Overall, the best combination of optimizations gives a mean speedup of 4.11 times compared to the same implementation without the optimizations.

Keywords: global cardinality constraint, constraint programming, global constraints, propagation algorithms

1. Introduction

Constraint programming is a powerful and flexible means of solving combinatorial problems. 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

Email address: `pn@cs.st-andrews.ac.uk` (Peter Nightingale) 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.

Consider a sports scheduling problem, where each team plays every other team exactly once in a season. No team can play two or more matches at the same time. Each team plays in a particular stadium at most twice during the season. In this example one might have two decision variables per match, representing the two teams. For a set of matches played in the same stadium, a global cardinality constraint [24] could be used to ensure no more than two occurrences of each team.

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 [34, 11].

The Global Cardinality Constraint (GCC) is a very important global constraint, present in various constraint solving toolkits, solvers and languages. It restricts the number of occurrences of values assigned to a set of variables. In the original version of the constraint [24], each value is given a lower bound and upper bound. In any solution, the number of occurrences of the value must fall within the bounds. The literature contains many propagation algorithms for this constraint, which trade strength of inference against computational cost, for example bound consistency [13, 19], range consistency [18], and generalized arc-consistency (GAC) [24, 18]. GCC is widely used in a variety of constraint models, for diverse problems such as routing and wavelength assignment [30], car sequencing [25], and combinatorial mathematics [11].

Returning to the sports scheduling example, GCC can be used to express the stadium constraint (that a team plays in a particular stadium at most twice during the season). Each value (representing a team) is given the bounds $(0, 2)$, and the variables are all slots at a particular stadium.

GCC has been generalized by replacing the fixed bounds on values with *cardinality variables* [18], where each cardinality variable represents the number of occurrences of a value. To avoid confusion, I refer to this as the Extended Global Cardinality Constraint (EGCC). Thus an EGCC constraint has *target* variables (where the number of occurrences of some values are constrained) and *cardinality* variables.

In this paper, I focus on the highest strength of inference (enforcing GAC) on the target variables. This allows the study of various methods in great depth, and leads to some surprising conclusions. I also survey methods for pruning the cardinality variables in depth. The main contributions of the paper are as follows.

- A literature survey of GAC propagation algorithms for the target variables, and their optimizations, in §3.

- Discussion of important implementation decisions in §3 that are frequently omitted from original papers, perhaps due to lack of space. For example, how to find augmenting paths for Régin's algorithm [24].

- The proposal of two new optimizations in §3.4. One of these is based on modifying the flow network of Régin's algorithm for greater efficiency, and the other is a novel generalization of the dynamic partitioning optimization of AllDifferent [6].
- A careful description of three concrete algorithms for pruning the cardinality variables in §4.
- Easily the largest empirical study of GAC propagation methods for the target variables of EGCC, in §5. This involves two basic algorithms and seven optimizations.
- Experimental conclusions and implementation advice for GAC for the target variables, in §6.
- An empirical study of pruning the cardinality variables, comparing the three methods, in §5.8, leading to experimental conclusions in §6.
- It is shown that an appropriate combination of optimizations is over 4 times faster on average than a careful but unoptimized implementation of Régin's algorithm (§5.10), for our benchmark set.
- A fast variant of EGCC is typically orders of magnitude better than a set of occurrence constraints. Even when EGCC propagation was least effective, it slowed the solver down by only 1.66 times or less in our experiments (§5.10).

2. Background

2.1. Preliminaries

A CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is defined as a set of n variables $\mathcal{X} = \langle x_1, \dots, x_n \rangle$, a set of domains $\mathcal{D} = \langle D(x_1), \dots, D(x_n) \rangle$ where $D(x_i) \subsetneq \mathbb{Z}$, $|D(x_i)| < \infty$ is the finite set of all potential values of x_i, and a conjunction $\mathcal{C} = C_1 \wedge C_2 \wedge \cdots \wedge C_e$ of constraints.

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

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

Although I define a constraint C_k to have scope $\langle x_{k_1}, \dots, x_{k_m} \rangle$, when discussing a particular constraint I frequently omit the k subscript, and refer to the variables as $\langle x_1, \dots, x_m \rangle$, and to the domains as $\langle D(x_1), \dots, D(x_m) \rangle$.

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

A *solution* to a CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ is a tuple τ of size $|X|$ where $\forall i : \tau[i] \in D(x_i)$ (τ represents an assignment to all variables), and all constraints are satisfied by τ: for each constraint C_k in $\mathcal{C}$ with scope $\langle x_{k_1}, \dots, x_{k_m} \rangle$, a new tuple τ' is constructed where $\forall j : \tau'[j] = \tau[k_j]$, and $\tau' \in C_k^S$ (τ' is acceptable).

Generalized Arc-Consistency (GAC) for constraint C_k is defined as a function from domains $\mathcal{D}_k$ to a set of literals P. Note that the set C_k^S is defined in terms of $\mathcal{D}_k$. A literal $x_i \mapsto j$ where $j \in D(x_i)$ is in P iff it is not present in any tuple in C_k^S: $\nexists \tau \in C_k^S : \tau[i] = j$. Literals in P are not part of any acceptable and valid tuple of the constraint, therefore they can be pruned without reducing the set of solutions of the CSP $\mathcal{P}$.

2.1.1. Graph theory

Régin's algorithm [24] and Quimper's algorithm [18] for pruning EGCC make use of network flow and bipartite matching theory [2] as well as strongly connected components [31]. Similarly, Régin's AllDifferent algorithm [23] makes use of results from graph theory, in particular maximum bipartite matching [1] and strongly connected components.

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 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.

A *digraph* $G = \langle V, E \rangle$ 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 direction.

2.1.2. Propagation and search

Propagation is one of the basic operations of most constraint solvers: it simplifies a CSP by pruning values from the domains. For example, applying GAC (defined above) to a constraint gives a set of values that may be pruned without changing the solution set. Constraint solvers provide a *propagation algorithm* (or *propagator*) for each type of constraint, and these are applied until the fixpoint is reached for all constraints.

Propagation is typically interleaved with splitting. Splitting is the basic operation of search, and a splitting operation transforms a CSP into two or more simpler CSPs. Hence a depth-first backtracking search is performed, with propagation occurring at each node in the search tree.

A propagator $\mathit{Prop}(C_k, \mathcal{D}_k)$ for constraint C_k computes a function from the domains $\mathcal{D}_k$ to new domains $\mathcal{D}'_k$. For example, the propagator may compute the GAC prunings P (defined above), and then prune each literal in P from $\mathcal{D}_k$ to construct $\mathcal{D}'_k$. Propagators only reduce variable domains (they are *contracting*): $\forall j : D'_{k_j} \subseteq D_{k_j}$. Propagators must also be correct with respect to C_k (the set C_k^S is preserved when the propagator is applied) and must not allow assignments that do not satisfy the constraint. These conditions (*correctness* and *weak monotonicity*) are defined by Schulte and Tack [28].

The propagators considered in this paper are *idempotent* (assuming that no variable is duplicated in $\langle x_{k_1}, \ldots, x_{k_m} \rangle$), which means that one application of the propagator will reach a fixpoint for the constraint: $\mathit{Prop}(C_k, \mathcal{D}_k) = \mathit{Prop}(C_k, \mathit{Prop}(C_k, \mathcal{D}_k))$.

2.2. Extended GCC

A traditional Global Cardinality Constraint has just one set of variables (the *target* variables). Each domain value has fixed lower and upper bounds associated with it. An assignment to the target variables is a solution iff the number of occurrences of each value is within the bounds for that value.

The main focus of this paper is the Extended Global Cardinality Constraint (EGCC). The EGCC has a second set of variables (*cardinality* variables) representing the number of occurrences of each value. Cardinality variables replace the fixed bounds on each value, hence EGCC is much more flexible than GCC. EGCC has the following form.

$$\mathrm{egcc}(X, V, C)$$

X is the vector of target variables, V is a vector of domain values of interest, and C is a vector of cardinality variables, one for each value in V. The constraint is satisfied under an assignment iff for all indices i of V, the number of variables in X set to V_i is equal to C_i. There is no restriction on the number of occurrences of any value not in V. (In Régin's original definition of GCC [24], each value in the target domains has a cardinality interval. In contrast, V might not include all values so a default interval of $0 \ldots \infty$ is used.) Throughout, I use r as the number of target variables $|X|$ for the constraint in question. I use d to represent the number of target variable domain values: $d = |D(x_1) \cup \ldots \cup D(x_r)|$ where $X = \langle x_1 \ldots x_r \rangle$.

Propagation of EGCC would typically be in two phases, to prune the target and cardinality variables respectively. Quimper et al [18] have shown that enforcing GAC on EGCC is NP-Hard in general. However it is known that when the domains of the cardinality variables are an unbroken interval then GAC is tractable [26]. To exploit this tractable case, the algorithms used in this paper read (and prune) only the bounds of the cardinality variables, and prune the target variables using only the bounds of the cardinality variables. The pruning of the target variables is similar to GAC (§2.1), however a new definition is required.

Definition 2.1. *For constraint* $C_k = \mathrm{egcc}(X, V, C)$ *with target variables* $X = \langle x_1 \ldots x_r \rangle$ *and cardinality variables* $C = \langle c_1 \ldots c_{|V|} \rangle$*, GAC-On-X is defined as a function from* $\mathcal{D}_k$ *to literal set* P *as follows. A new constraint* $C'_k =$

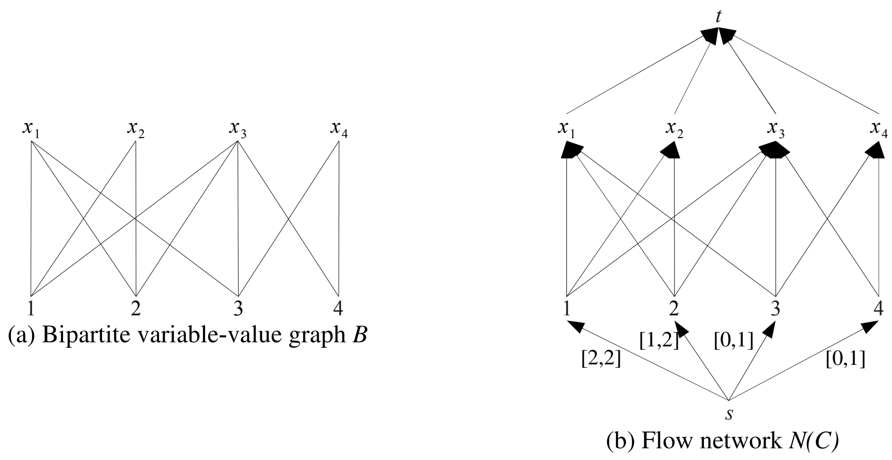


Figure 1: Example of variable-value graph and flow network

egcc(X, V, C') *is constructed with* $C' = \langle c'_1 \dots c'_{|V|}\rangle$, *and domains* $\forall i : D(c'_i) = \{\underline{c_i} \dots \overline{c_i}\}$. *The GAC function is applied to* C'_k *to obtain literals* P'. *Finally* P *is the set of literals in* P' *pertaining to* X: $P = \{(y_i \mapsto a) \in P' \mid y_i \in X\}$.

GAC-On-X for an EGCC constraint is equivalent to reading the bounds of the cardinality variables, creating a new GCC constraint with those bounds, and enforcing GAC on the GCC.

Samer and Szeider identify other tractable cases, for example when the treewidth of the variable-value graph is bounded [26]. While this work is of theoretical interest, it is not clear that the tractable cases would be found in typical uses of the EGCC constraint.

2.3. Basic definitions for EGCC

I will refer to the target variables X as $x_1, \dots, x_r$ and their domains as $D(x_1), \dots, D(x_r)$. The size of the union of all target domains is d. For simplicity, domain elements are assumed to be $1 \dots d$.

First the *variable-value graph* is defined. The variable-value graph has one set of vertices representing target variables, and a second set representing values. There is an edge between a variable x_i and a value a iff $a \in D(x_i)$. Figure 1(a) gives an example of a variable-value graph.

Definition 2.2. *Given an EGCC* K, *the bipartite variable-value graph is defined as* $B(K) = \langle V, E\rangle$ *where* $V = \{x_1, \dots, x_r, 1, \dots, d\}$ *and* $E = \{x_i \leftrightarrow j \mid j \in D(x_i)\}$

Next a flow network $N(K)$ for an EGCC K is defined. It is derived from the variable-value graph. $N(K)$ has both a capacity c and lower bound l on each edge. It includes the vertices in the variable-value graph, and also a source vertex s and a sink t. It is defined below and an example is given in Figure 1(b).

Definition 2.3. *Given an EGCC K with parameters $X = \langle x_1 \dots x_r \rangle$, $V = \langle v_1 \dots v_m \rangle$, and $C = \langle c_1 \dots c_m \rangle$*[2]*, the flow graph $N(K)$ is defined as a digraph $N(K) = \langle V, E \rangle$ where $V = \{x_1, \dots, x_r, 1, \dots, d, s, t\}$. E is the union of the following edge sets.*

- *For each edge in $B(K)$, orient the edge from values to variables. For all edges (v, x) in this set $l(v, x) = 0$ and $c(v, x) = 1$.*
- *For all value vertices $v_i \in V$, there is an edge (s, v_i) with lower bound $l(s, v_i) = \underline{c_i}$ and capacity $c(s, v_i) = \overline{c_i}$ (i.e. the flow through (s, v_i) is within the bounds of the cardinality variable c_i).*
- *For all values a in $\{1 \dots d\}$ but not in V, there is an edge (s, a) with $l(s, a) = 0$ and $c(s, a) = \infty$.*
- *For all variables x_i, there is an edge (x_i, t) where $l(x_i, t) = 0$ and $c(x_i, t) = 1$.*

The intuition behind $N(K)$ is that an integer flow from s to t corresponds to an assignment to the target variables. If the flow uses an edge (x_i, a) then in the assignment, $x_i = a$. If a flow in $N(K)$ covers all the variable vertices, and meets all the lower bounds and capacities, it corresponds to a satisfying assignment to the target variables.

2.4. Hall sets and EGCC

Hall sets are useful for understanding the pruning of the target variables in EGCC. Two types of Hall set are required, for the upper bounds and lower bounds respectively. The following definition of upper-bound Hall set is equivalent to Quimper's definition (see [17] §5.1).

Definition 2.4. *A UB-Hall set H_u is a set of variables with corresponding values $D(H_u) = \bigcup\{D(x_i) \mid x_i \in H_u\}$ such that the sum of the upper bounds of $D(H_u)$ equals the number of variables: $|H_u| = \sum_{v_i \in D(H_u)} \overline{c_i}$.*

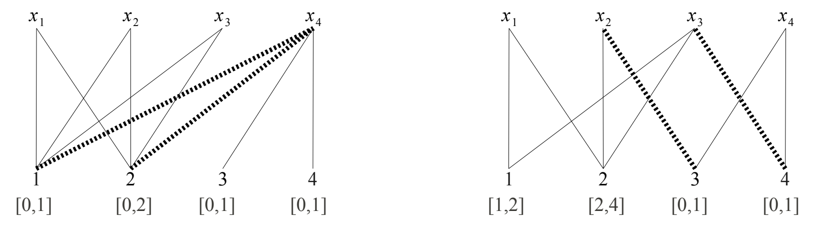
In any solution to the constraint, the variables H_u are assigned to values in $D(H_u)$ and this assignment meets the upper bound for each value in $D(H_u)$. Therefore no other variable $x_j \notin H_u$ can be assigned a value in $D(H_u)$, and some pruning may be performed. Variables H_u *consume* the values $D(H_u)$.

A small example of a UB-Hall set is given in Figure 2(a). In this case, three variables $\{x_1, x_2, x_3\}$ are adjacent to only the values $\{1, 2\}$. The sum of the upper bounds of $\{1, 2\}$ is three, therefore $\{x_1, x_2, x_3\}$ is a UB-Hall set.

For lower bounds, the Hall set is similar but variables and values are swapped. This definition is equivalent to *unstable sets* as defined by Quimper (see [17] §5.2).

[2]For simplicity it is assumed that $V \subseteq \{1, \dots, d\}$. If this is not the case, for each value v_i not in $\{1, \dots, d\}$ the corresponding cardinality variable c_i is set to 0.

GCC instances represented as variable-value graphs. Values are labelled with their lower and upper bound as an interval [*a*,*b*].



(a) Example of a UB-Hall set.
Variables {*x*1, *x*2, *x*3} consume values {1,2}.
The dotted lines will be pruned.

(b) Example of an LB-Hall set.
Values {1,2} consume variables {*x*1, *x*2, *x*3}.
The dotted lines will be pruned.

Figure 2: Example of a UB-Hall set and an LB-Hall set

Definition 2.5. *An LB-Hall set H_l is a set of values with corresponding variables* $\text{Vars}(H_l) = \{x_i \mid H_l \cap D(x_i) \neq \emptyset\}$ *such that the sum of the lower bounds of H_l equals the number of variables:* $|\text{Vars}(H_l)| = \sum_{v_i \in H_l} \underline{c_i}$.

In this case, in any solution to the constraint, the variables $\text{Vars}(H_l)$ must be assigned to values in H_l exclusively, to meet the lower bounds of H_l. Therefore other values may be pruned. The values H_l *consume* the variables $\text{Vars}(H_l)$.

A small example of an LB-Hall set is shown in Figure 2(b). The sum of the lower bounds for values $\{1, 2\}$ is three, and the two values are adjacent to three variables $\{x_1, x_2, x_3\}$, therefore $\{1, 2\}$ is an LB-Hall set. This leads to the pruning of two values.

The definition of LB-Hall set captures the reason for prunings but not failure. The constraint fails ($C_k^S = \emptyset$) if there exists a set of values H_l where the sum of the lower bounds is greater than the number of variables: $|\text{Vars}(H_l)| < \sum_{v_i \in H_l} \underline{c_i}$.

UB and LB-Hall sets are closely related to enforcing GAC-On-X (Definition 2.1). Quimper [17] proved that lower bounds and upper bounds can be considered separately without losing GAC-On-X (thus decomposing EGCC into an upper-bound constraint (ubc) and a lower-bound constraint (lbc)). He also showed the correspondence between Hall's marriage theorem and the satisfiability of the ubc. It follows that finding all UB-Hall sets H_u and pruning values in $D(H_u)$ from other variables is sufficient to enforce GAC-On-X on the ubc. For lbc, Quimper shows directly that finding all LB-Hall sets is sufficient to enforce GAC-On-X.

The algorithms presented in the next section make use of UB- and LB-Hall sets to prune the target variables.

2.5. Experimental context

Experiments were performed with Minion [4, 5] version 0.9. The solver was modified only to add variants of EGCC. In this section I give an overview of Minion.

Constraint solvers provide a propagation loop that calls propagators until the global fixpoint is reached. Propagators subscribe to variable events and are scheduled to be executed when one of the events occurs. Subscription to an event is referred to as *placing a trigger*, where a trigger is an object and it is placed into a list related to the event. A propagator is *triggered* when it is called because an event occurred. Minion provides the following variable event types: $\max(D(x_i))$ changed; $\min(D(x_i))$ changed; value a removed from $D(x_i)$; $D(x_i)$ changed in any way; x_i is assigned.

Triggers are identified by a number which is passed to the propagator. Therefore a propagator can identify the exact event that caused it to be called. Notification of the events is important for several of the EGCC propagators; without this facility the propagator would scan the variable domains, adding a linear or quadratic cost. The exact use of variable events is described in §3.5.2.

Minion is a variable-centric solver with an additional constraint-centric queue. 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 (whereas with the constraint queue, each trigger is copied to the queue). However, the variable queue does not allow constraints to be given different priorities. Having the additional constraint queue overcomes this limitation.

The variable queue contains the variable events listed above. The constraint queue contains pointers to constraints. Constraints are responsible for adding themselves to the constraint queue as necessary. It 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, only EGCC and AllDifferent constraints use the constraint queue.

Propagators may require internal state for efficiency. Minion provides both backtracked memory (that is restored as search backtracks) and non-backtracked memory. The backtracked memory must be allocated before search begins, and is blocked together. It is backtracked by copying the block. The consequences of this memory architecture are discussed in §5.2.1.

3. Pruning the target variables of EGCC

In this section I discuss pruning the target variables, beginning with a survey of the relevant literature. There are two published algorithms to enforce GAC-On-X, given lower and upper bounds for the occurrence of each value. Régin [24] presented an algorithm based on network flow. It makes use of the Ford-Fulkerson algorithm [2] to compute a flow which represents an assignment to the target variables. The assignment satisfies the lower and upper bounds for each value. Then Tarjan's algorithm is used to compute the set of edges that cannot belong to any maximum flow. These edges correspond to domain values to be pruned. The time complexity of one call to the algorithm is $O(r^2d)$, dominated by the Ford-Fulkerson algorithm.

An alternative algorithm was presented by Quimper et al [18, 17][3]. The approach here is to split the GCC into two constraints, such that enforcing GAC on both is equivalent to enforcing GAC on the GCC. In this way they obtain a better time bound than Régin's algorithm.

3.1. Régin's algorithm

The first stage of Régin's algorithm computes a flow that is both *feasible* (it meets all lower bounds) and maximum, without exceeding capacities. First a feasible flow is computed, then it is extended to a maximum flow.

3.1.1. Computing a feasible flow

To compute a feasible flow, a second flow network $LB(K)$ is used that is identical to $N(K)$ with one additional edge. There is an edge (t, s) with $l(t, s) = 0$ and $c(t, s) = \infty$.

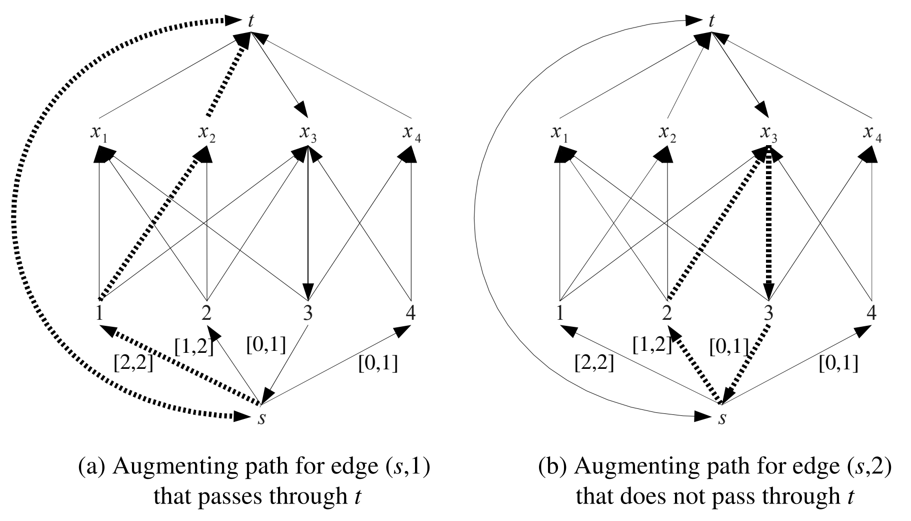
In order to use the Ford-Fulkerson algorithm [2], Régin defines the *residual graph* for a flow network and flow. A flow is a function mapping all edges to the quantity of material passing through them (a non-negative integer). The intuition behind the residual graph is that there is an edge from vertex a to vertex b iff it is possible to increase the flow from a to b without violating the capacity $c(a, b)$, or to reduce the flow from b to a without violating the lower bound $l(b, a)$. (The first case applies when (a, b) is an edge in $N(K)$, and the second case applies when (b, a) is an edge in $N(K)$).

Definition 3.1. *The residual graph $Res(G, f)$ is derived from a flow network G and a flow f. It is a digraph with the same set of vertices as G. For each edge (a, b) in G, if $f(a, b) > l(a, b)$ then the edge (b, a) is present in $Res(G, f)$. If $f(a, b) < c(a, b)$ then the edge (a, b) is present in $Res(G, f)$. No other edges are present in $Res(G, f)$.*

The algorithm to compute a feasible flow is as follows. Suppose f is an infeasible flow. Pick an edge (a, b) from $LB(K)$ such that $f(a, b) < l(a, b)$. Find a simple path from b to a in $Res(LB(K), f)$. This is named an augmenting path, and (by the definition of $Res(LB(K), f)$) the flow can be increased along this path and through (a, b) by 1 unit. This is denoted *applying* the augmenting path. For each edge (x, y) in the path, either $f(x, y)$ is increased or (if the edge is oriented (y, x) in $LB(K)$) $f(y, x)$ is decreased. This creates a new flow f' where $f'(a, b) > f(a, b)$. In this context, the increase in the flow through (a, b) is always 1. If there is no augmenting path from b to a, then it is impossible to satisfy the lower bound and the EGCC fails.

Figure 3 shows two examples of augmenting paths in the residual graph $Res(LB(K), f)$. The existing flow f passes through $(s, 3)$, $(3, x_3)$, (x_3, t), and (t, s). In CSP terms, this flow represents the assignment $x_3 = 3$.

[3]The algorithm was described in Claude-Guy Quimper's PhD thesis [17] therefore I refer to it as Quimper's algorithm throughout.



t
x1
x2
x3
x4
1
2
3
4
s
[2,2]
[1,2]
[0,1]
[0,1]
(a) Augmenting path for edge (s,1) that passes through t
(b) Augmenting path for edge (s,2) that does not pass through t

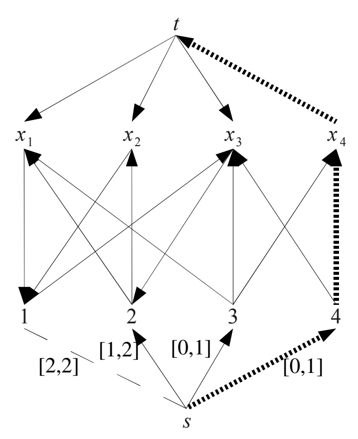
Figure 3: Examples of augmenting paths in the residual graph to compute a feasible flow

For EGCC, the only edges where the lower bound is non-zero are those from s to a domain value, (s, v_i). An augmenting path has one of two forms. It passes through the edge (t, s) as shown in Figure 3(a), or it does not pass through (t, s) as shown in Figure 3(b). In the first case, applying the augmenting path increases the overall flow from s to t (by assigning x_2 to 1 in this example). In the second case, applying the augmenting path does not affect the overall flow from s to t. In this example, the flow through $(s, 2)$ is increased (by setting x_3 to 2) and the flow through $(s, 3)$ is decreased.

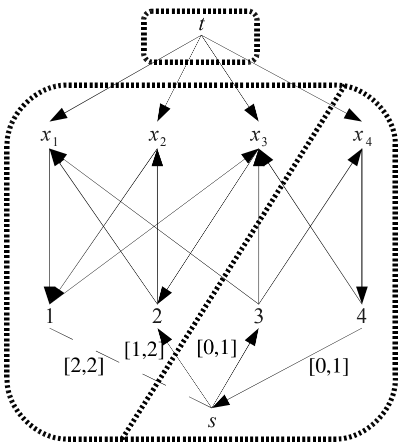
For this paper the implementation iterates through the values v_i where $f(s, v_i) < l(s, v_i)$ and meets the lower bound for v_i if possible. An augmenting path is sought starting at the vertex v_i. The search succeeds when it discovers s or t or fails when all reachable vertices have been explored. Terminating at s corresponds to Figure 3(b). When terminating at t, the edge (t, s) is appended to the augmenting path and this corresponds to Figure 3(a).

3.1.2. Computing a maximum flow from a feasible flow

Given a feasible flow f_0, the Ford-Fulkerson algorithm is used again to compute a maximum feasible flow. An augmenting path is sought from s to t in $Res(N(K), f_0)$. This is applied to create flow f_1. The process is repeated for f_1 to create f_2, etc. The algorithm terminates when no augmenting path exists from s to t in $Res(N(K), f_k)$. If the maximum feasible flow f_k does not cover all variable vertices, then the constraint fails. An example is given in Figure 4(a). In this example, the feasible flow f_0 uses edges $(1, x_1)$, $(1, x_2)$ and $(2, x_3)$, therefore these edges are reversed in $Res(N(K), f_0)$. The augmenting path uses edge $(4, x_4)$ and completes the maximum flow.



(a) Augmenting path from s to t



(b) SCCs of the residual graph

Figure 4: Examples of computing a maximum feasible flow and the SCCs of the residual graph

3.1.3. Finding augmenting paths in Ford-Fulkerson

The two main options here are depth-first search (FF-DFS) and breadth-first search (FF-BFS). The problem is very similar to maximum bipartite matching: an augmenting path alternates between variables and values (ignoring s and t). Therefore I refer to the bipartite matching literature.

Setubal empirically compared ABMP, FF-BFS, FF-DFS and Goldberg's algorithm [29]. He generated bipartite graphs with 2^p vertices in each partition, where $p \in \{8 \dots 17\}$. With an estimate of 2^9 vertices or fewer in each partition[4], an examination of Setubal's results on sequential computers (taking the size closest to 2^9 and all smaller sizes) shows that FF-BFS is competitive for all classes and is most efficient (or equal) in 8/11 classes of graphs, and 10/13 sets of a particular size. Setubal recommends using FF-BFS for graphs up to thousands of vertices. Given these results I used FF-BFS throughout.

3.1.4. Pruning the domains

The second stage of Régin's algorithm makes use of *strongly connected components* (SCCs). An 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 $O(|V| + |E|)$ time [31].

An edge of the form (v_i, x_j) from $N(K)$ that cannot be in *any* maximum

[4]The largest EGCC constraint in the benchmark instances has 200 variables and fewer values, so they are all smaller than 2^9.

feasible flow corresponds to a value to be pruned (i.e. v_i is pruned from x_j). Given the maximum feasible flow f that covers all variable vertices, the residual graph $Res(N(K), f)$ is partitioned into its SCCs. If an edge (v_i, x_j) goes between two SCCs *and* is not used in the flow f, then the algorithm prunes v_i from $D(x_j)$.

The intuition behind this result is that for any edge (v_i, x_j) in the residual graph, if v_i and x_j are in the same SCC then there is a simple path from x_j to v_i (by the definition of SCCs). This path may be used as an augmenting path to increase the flow through (v_i, x_j). Hence (v_i, x_j) can take part in a maximum feasible flow. However, if v_i and x_j are in different SCCs, there is no path from x_j to v_i. (v_i, x_j) cannot take part in any maximum feasible flow, unless it is in f. In this case the algorithm prunes v_i from x_j.

Another understanding of Régin's algorithm comes from Hall sets. Every pruning is justified by a UB-Hall set (def. 2.4) or an LB-Hall set (def. 2.5); see §2.4. For a deletion of a from $D(x_i)$, either a is consumed by a UB-Hall set that does not contain x_i, or x_i is consumed by an LB-Hall set that does not contain a. In both cases, the Hall set corresponds directly to an SCC of the residual graph: in the first case, the SCC containing a; in the second the SCC containing x_i.

Figure 4(b) shows an example where f flows through edges $(1, x_1)$, $(1, x_2)$, $(2, x_3)$ and $(4, x_4)$. SCCs of $Res(N(K), f)$ are marked with thick dotted lines. The edges $(4, x_3)$, $(3, x_3)$ and $(3, x_1)$ cross between SCCs, so the corresponding values are pruned from the target domains. Tarjan's algorithm and the pruning of domains is implemented exactly as described in [6].

3.1.5. Time complexity of Régin's algorithm

If δ is the number of edges in $B(K)$ (i.e. the sum of the sizes of target variable domains), and r is the number of target variables, the time to find a maximum feasible flow with Ford-Fulkerson is $O(r\delta)$. (No more than r augmenting paths are found and applied.) The complexity of Tarjan's algorithm is $\Theta(\delta)$ (i.e. run time is bounded above and below by δ asymptotically), because Tarjan's algorithm uses every edge in the graph.

Régin suggests that Dinic's algorithm [32] should be faster in practice than Ford-Fulkerson [24]. However Dinic's algorithm with the Sleator-Tarjan method of finding a blocking flow (as described by Tarjan [32]) has an upper bound of $O(r\delta \log(r+d))$. (This bound may not be tight for our problem.) In this paper I do not consider Dinic's algorithm because of its greater complication and worse time bound.

3.2. Quimper's algorithm in detail

The approach taken by Quimper et al [18, 17] is to split the GCC into a lower bound constraint (*lbc*) and an upper bound constraint (*ubc*). The *lbc* ensures that the lower bound for each value is respected, and similarly the *ubc* enforces the upper bound. Enforcing GAC on the *lbc* and *ubc* independently prunes the same values as GAC on the GCC [19]. For both the *lbc* and *ubc*, a two-stage algorithm similar to Régin's is used.

For the first stage (of both *lbc* and *ubc*), the variable-value graph $B(K)$ is used, and a structure similar to a maximum matching is computed. A conventional maximum matching M is a maximum-cardinality set of edges of $B(K)$ such that no vertex occurs more than once in M. This is generalized by allowing some vertices to occur more than once: value vertices may occur multiple times up to a capacity $\text{cap}(a)$ for a value a. Variable vertices occur at most once.

A generalized maximum matching is computed using a modified Hopcroft-Karp algorithm [10]. The modification is very simple and does not affect the worst-case execution time. In the *lbc* $\text{cap}(a)$ is set to the lower bound of a for each value a, and for *ubc* the upper bound is used. At this point, the *lbc* fails if the matching does not meet all the lower bounds. The *lbc* matching is completed (to cover all variable vertices) by matching each unmatched variable vertex with an arbitrary value. The *ubc* fails if the generalized matching does not cover all variables.

For the second stage of both algorithms, the matchings are translated to flows in $N(K)$. For a matching M and corresponding flow f_M, each edge $(x, y) \in M$ carries a unit of flow in f_M. Each edge from $B(K)$ not in M carries no flow in f_M.

The second stage of Régin's algorithm is used with changes to the bounds: for the *ubc* 0 is used as the lower bound for all values; and for the *lbc* ∞ is used as the upper bound for all values.

Finally, it is not necessary to run the two propagators alternately to a fixpoint to enforce GAC on the GCC. It is sufficient to run one then the other. The implementation used in this paper runs the *lbc* propagator then the *ubc* propagator.

The use of Hopcroft-Karp in place of Ford-Fulkerson produces a tighter time bound of $O(r^{1.5}d)$ (or $O(r^{0.5}\delta)$) for one call to the propagator. Although Quimper's algorithm has a tighter upper bound, it may not be better in practice because it maintains two maximal matchings rather than one in Régin's algorithm, and makes two calls to Tarjan's algorithm rather than one. The two algorithms are compared experimentally in §5.3.

3.3. Review of optimizations of the basic algorithms

The algorithms described above are similar to each other and to Régin's AllDifferent algorithm [23]. A number of optimizations to this collection of algorithms have been proposed by various authors. They are surveyed in this section.

3.3.1. Incremental matching

The maximum flow M (or matchings M_l and M_u in the case of Quimper's algorithm) may be maintained incrementally during search [24]. This is done by storing M between calls to the propagator. When the propagator is called, M may no longer be maximum because of domain removals, so the flow or matching algorithm is used to repair it. For AllDifferent, incremental matching has been shown to improve efficiency [6].

3.3.2. Incremental graph maintenance

The original GAC AllDifferent algorithm [23] stores its graph between calls, maintaining the graph incrementally as variable domains change. One parameter of the algorithm is the set of values deleted from variable domains, and the first step of the algorithm is to update the graph. This idea has two costs: updating the graph by removing edges; and backtracking the graph as search backtracks. Whether the benefit outweighs the cost is an empirical question which is answered below. The implementation of incremental graph maintenance is discussed in §3.5.1.

An algorithm without incremental graph maintenance can discover the graph as it is traversed, by querying variable domains and the maximum flow. This is the approach used for AllDifferent by Gent et al [6].

3.3.3. Priority queue

Many constraint solvers have a priority queue for constraints (e.g. Choco [14], Gecode [27]), such that the priorities determine the order in which constraint propagators are executed. It is standard practice for the EGCC to have a low priority. Schulte and Stuckey demonstrate the importance of priority queueing [27], and it is evaluated in the experiments here.

3.3.4. Staged propagation

Schulte and Stuckey proposed multiple or staged propagation for AllDifferent [27], where a cheap propagator with a high priority is combined with a more expensive, low priority propagator.

I do not experiment with staged propagation for EGCC in this paper, however it would be an interesting area for future work.

3.3.5. Dynamic Partitioning

Gent et al [6] proposed an algorithm which partitions an AllDifferent constraint during search. Suppose for example we have AllDifferent($x_1 \dots x_6$) and have $x_1 \dots x_3 \in \{1 \dots 3\}$, $x_4 \dots x_6 \in \{4 \dots 6\}$. This can be partitioned into two independent cells: AllDifferent($x_1 \dots x_3$) and AllDifferent($x_4 \dots x_6$). The main benefit is that if some variable x_i has changed, the propagator need only be executed on the cell containing x_i, not the original constraint. This saves time in Tarjan's algorithm.

A cheap way of obtaining the partition is to use the SCCs of the residual flow network, which are computed as part of Régin's AllDifferent algorithm. In some cases it is possible to find a finer partition than the SCCs. However, experiments showed that using SCCs as the partition is effective in practice [6].

In this paper I generalize dynamic partitioning to the EGCC constraint. This is described in §3.4.2.

3.3.6. Assigned variable removal

The implementation of EGCC in Gecode [27] updates its array of target variables each time it is called, removing assigned variables[5]. This promises to be a lightweight and effective optimization. It is evaluated in §5.6.

Dynamic partitioning subsumes assigned variable optimization, because any assigned variable is a singleton SCC and therefore cannot be in any active cell of the constraint. However, dynamic partitioning is likely to be more expensive.

3.3.7. Domain Counting

Recall that Quimper's algorithm divides the constraint into the upper-bound constraint (ubc) and lower-bound constraint (lbc). Quimper and Walsh observed that the ubc need not be propagated when domains are large [20]. They proposed an algorithm that constructs a sorted list of the sizes of all target variable domains. It iterates through the list and determines whether the ubc propagator is run. I suspect this algorithm would be too expensive for general use, although on some problem classes it may prove valuable. Quimper and Walsh do not give a domain counting algorithm for the lbc.

A simpler form of domain counting has been used for AllDifferent. Lagerkvist and Schulte used the following scheme: when triggered by a target variable x_i the propagator only runs if $|D(x_i)| \leq r$ (where r is the arity of the constraint) [15]. Gent et al improved the threshold to $|D(x_i)| \leq r - 1$ [6], but did not find domain counting to be useful in experiments.

It is possible to derive a similar domain size threshold for the ubc, using the definition of a UB-Hall set (definition 2.4). However it is not possible for the lbc. Consider the definition of an LB-Hall set (definition 2.5). The size of the domains of the variables in the LB-Hall set is not restricted by the definition. Since it is not possible for the lbc, it is not possible for the EGCC. I do not experiment with domain counting in any form.

3.3.8. Important Edges

Katriel observed that many value removals affecting a GCC constraint result in no other value removals, and so work processing them is wasted [12]. She introduces the concept of an *important edge* of the residual graph. An important edge is one whose removal causes the pruning of some variable-value pair. Therefore, when an *un*important edge is removed, it is not necessary to run the propagator.

Where r is the number of target variables, Katriel gave an upper bound of $3r$ on the number of important edges that correspond to domain values (i.e. edges between variable vertices and value vertices).

Katriel shows that if there are many allowed values per variable, the expected cost of propagation can be reduced. She proposes to keep a count of pruned values, and run the propagator only when the counter reaches a threshold value. The threshold is set so that the propagator is likely to prune a value when

[5]Guido Tack, personal communication

executed. This algorithm does not always enforce GAC on the GCC. 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 probabilistic algorithm would be interesting, the fact that it does not maintain GAC on the GCC puts it outside the scope of this paper.

Gent et al [6] gave an algorithm for AllDifferent to identify a small set of edges containing the important edges and possibly others. The identified edges correspond to important domain values. The propagator is only executed when an important domain value has been removed, thus maintaining GAC with fewer calls to the propagator. This approach is adapted for EGCC in §3.5.3.

3.3.9. Entailment

Quimper et al give the conditions under which the GCC constraint is *entailed* (i.e. there are no unacceptable tuples in the relation of the constraint, under the current domains) [18]. If the constraint is entailed, it need not be propagated at the current search node or its descendents. For the lower bound constraint, the condition is that for each value v with lower bound LB(v), LB(v) variables are assigned to v. Similarly, for the upper bound constraint, for each value v, at most UB(v) domains contain v. However, EGCC cannot be entailed until all variables are assigned. If some variable is not assigned, any acceptable tuple may be turned into an unacceptable tuple by changing the value of that variable.

I did not experiment with entailment of GCC because the conditions are quite tight, and are likely to occur only when a large number of variables are assigned, therefore the benefit appears to be limited. Also the architecture of Minion is not well suited to entailment (as discussed in §5.2.1).

3.4. Novel optimizations for pruning the target variables

In this section I describe two optimizations. The first is a change to Régin's algorithm intended to improve the computation of a maximum flow. The second generalizes dynamic partitioning (described in §3.3.5) to EGCC.

3.4.1. Transpose graph for computing the maximum flow

To compute a maximum feasible flow from a feasible flow, Régin's algorithm uses the graph $N(K)$, and seeks paths from s to t in $N(K)$. An alternative would be to use the transpose of $N(K)$ denoted $N(K)^T$. The transpose is $N(K)$ with the direction of every edge reversed. A path from t to s in $N(K)^T$ is equivalent to a path from s to t in $N(K)$.

The direction of the flow f is reversed to form f^T, and the algorithm searches for paths from t to s in the residual graph $Res(N(K)^T, f^T)$. The algorithm works as follows. Iterate through edges (t, x_i) that carry no flow. For each edge, search for a path p from x_i to s. If there is such a path, augment the flow along p and through (t, x_i). If there is no path p, it is not possible to construct a flow that covers all variables so the algorithm fails immediately.

The conventional Régin's algorithm completes the maximum flow before testing if it covers all variables. When using the transpose graph, the algorithm can potentially stop much earlier, when it discovers a variable that cannot take part in a maximum flow. Also, each search for an augmenting path is more focused since it starts with a specific variable. Quimper's algorithm uses the transpose graph, however (like Régin's algorithm) it completes the maximum flow before testing if it covers all variables [18].

In §5.4 this approach is evaluated compared to Régin's original algorithm. For both algorithms, a breadth-first search is used to find augmenting paths.

3.4.2. Dynamic Partitioning

Dynamic partitioning essentially re-writes the EGCC constraint into multiple independent constraints as domains are narrowed. As described in §3.3.5, Gent et al gave an algorithm for dynamic partitioning of AllDifferent [6]. The AllDifferent algorithm maintains a partition of the set of variables. I generalize the algorithm to EGCC. Consider the following EGCC constraint.

$$\begin{aligned}&x_1 \dots x_3 \in \{1,3\}, x_4 \dots x_6 \in \{2,3,4\},\\ &c_1, c_2 \in \{0,1\}, c_3, c_4 \in \{0,1,2\},\\ &\text{EGCC}([x_1 \dots x_6],[1,2,3,4],[c_1,c_2,c_3,c_4])\end{aligned}$$

GAC-On-X propagation removes value 3 from variables $x_4 \dots x_6$. Following this, the domains of $x_1 \dots x_3$ and $x_4 \dots x_6$ are disjoint, and the constraint can be re-written into two constraints as shown below.

$$\begin{aligned}&\text{EGCC}([x_1 \dots x_3],[1,3],[c_1,c_3])\\ &\text{EGCC}([x_4 \dots x_6],[2,4],[c_2,c_4])\end{aligned}$$

Suppose x_3 were assigned to 3. The first of the two constraints can be re-written again as follows.

$$\begin{aligned}&\text{EGCC}([x_1, x_2],[1,3],[c_1,(c_3-1)])\\ &\text{EGCC}([x_3],[3],[1])\\ &\text{EGCC}([x_4 \dots x_6],[2,4],[c_2,c_4])\end{aligned}$$

In this case, the domains of $[x_1, x_2]$ and x_3 are not disjoint, they share the value 3. One occurrence of 3 resides with x_3, and $c_3 - 1$ occurrences of 3 reside with $[x_1, x_2]$. Suppose x_1 were also assigned 3. Now the occurrences of 3 have reached its upper bound, so after propagation and further re-writing we have this situation.

$$\begin{aligned}&\text{EGCC}([x_2],[1],[1])\\ &\text{EGCC}([x_1],[3],[1])\\ &\text{EGCC}([x_3],[3],[1])\\ &\text{EGCC}([x_4 \dots x_6],[2,4],[c_2,c_4])\end{aligned}$$

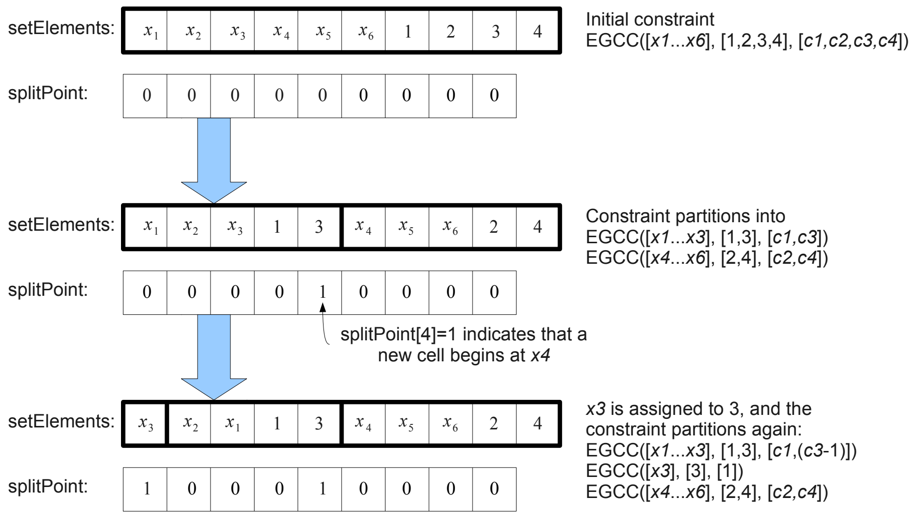

setElements:
x_1 x_2 x_3 x_4 x_5 x_6 1 2 3 4
Initial constraint
EGCC([x1...x6], [1,2,3,4], [c1,c2,c3,c4])
splitPoint:
0 0 0 0 0 0 0 0 0
setElements:
x_1 x_2 x_3 1 3 x_4 x_5 x_6 2 4
Constraint partitions into
EGCC([x1...x3], [1,3], [c1,c3])
EGCC([x4...x6], [2,4], [c2,c4])
splitPoint:
0 0 0 0 1 0 0 0 0
splitPoint[4]=1 indicates that a new cell begins at x4
setElements:
x_3 x_2 x_1 1 3 x_4 x_5 x_6 2 4
x3 is assigned to 3, and the constraint partitions again:
EGCC([x1...x3], [1,3], [c1,(c3-1)])
EGCC([x3], [3], [1])
EGCC([x4...x6], [2,4], [c2,c4])
splitPoint:
1 0 0 0 1 0 0 0 0

Figure 5: The refinement of the partition as an EGCC constraint is re-written as multiple constraints.

The EGCC algorithm maintains a partition of the set containing target variables and values. The major changes from AllDifferent are that values are included in the partition, and corner cases of EGCC (involving singleton variables and values) are accounted for. Initially the partition has one cell, consisting of all target variables and values. The partition is refined as propagation and search progresses, and restored as search backtracks. In the example above, the final refined partition would be $\{\{x_1\}, \{x_2\}, \{x_3\}, \{1\}, \{3\}, \{x_4, x_5, x_6, 2, 4\}\}$. Assigned variables are singleton sets, and so are values where the number of occurrences has reached the upper bound.

The partition I use corresponds to the SCCs of the residual graph (§3.1.4), and these are stored in the partition data structure described in [6]. (Target variables are represented using integers $0..r-1$, and values using $r..r+d-1$ if there are d values.) The data structure allows an item to be located in $O(1)$ time, and its cell to be iterated in linear time. Splitting a cell also takes linear time, and undoing the split operation on backtracking is $O(1)$.

Figure 5 gives an example of how the partition data structure of [6] works on EGCC. Each cell is stored in setElements in a contiguous block in no particular order. The array splitPoint marks where a cell ends. Only splitPoint is backtracked as search backtracks, hence cells join back together but the elements may be in a different order. (A third array maps a variable or value to its index in setElements, hence allowing it to be located in $O(1)$ time.)

An assigned target variable forms a singleton SCC, therefore the assigned variables are removed from the active cells of the constraint and cause almost no overhead.

Triggering with Dynamic Partitioning

The constraint maintains a set τ of target variables and values to be processed. When the constraint is notified of a domain change event, it adds the variable changed (for target variables) or the corresponding value (for cardinality variables) to τ. τ is cleared after the propagator executes and whenever search backtracks.

When the propagator is called, it iterates through τ and constructs a set of the cells to be propagated. A cell is propagated iff the cell contains a variable or value in τ. Propagation is performed on each cell in this set independently. Cells that are not propagated are almost cost-free. This scheme relies on the solver notifying the propagator of changed variables. If this information were not available, the propagator could discover the changed variables by iterating through each target variable domain, however this would add a quadratic cost and may outweigh any speed-up caused by the optimization.

Dynamic partitioning affects the worst-case analysis of Tarjan's algorithm. Without dynamic partitioning, the bound is $\Theta(\delta)$, where δ is the number of edges in the residual graph. With dynamic partitioning, the bound is $O(\delta)$ because it only runs Tarjan's algorithm on triggered cells of the constraint, in effect ignoring parts of the residual graph.

3.5. Implementation of optimizations from the literature

In this section I describe the implementation details of optimizations found in the literature, when these are not specified in the original papers.

3.5.1. Incremental graph maintenance

In this optimization, the variable-value graph is stored between calls and updated incrementally. This was first used by Régin [23] and is described in Section 3.3.2. For each vertex in the variable-value graph, an iterable list of adjacent vertices is required. The order of iteration is not important, but obtaining the next element should be $O(1)$. Similarly removing an element and testing its presence in the list should be $O(1)$ operations. Restoring the list on backtracking should be as cheap as possible.

The following representation is used, where each vertex is represented by a unique integer from a small range.

List An array of vertices (integers), not backtracked.

ListSize A single integer representing the size of the adjacency list. This must be backtracked.

InvList An array mapping vertices to their positions in List. Not backtracked.

This representation has the advantages of minimizing the backtracking memory and being directly iterable. The removal operation for a vertex a is to swap it with the item at the end of the list (i.e. place it in position ListSize-1), and then to reduce ListSize by one, thus a disappears from the list. On backtracking, ListSize is restored and a reappears in the list, at the end. InvList allows a

to be found in constant time, and is updated when the swap is performed. Thus the removal operation is $O(1)$. An item a is in the list iff InvList$[a]$ < ListSize. This data structure is used by the solver Mistral [8]. ListSize is backtracked by copying, as described in §2.5.

The constraint is notified of each pruned value for all target variables. These events are used to maintain the adjacency lists and queue the constraint for propagation if necessary.

Fixpoint reasoning

It may be helpful to perform fixpoint reasoning [27]. The EGCC propagator is idempotent if there are no repeated variables. When it prunes a value from a target variable, it will be notified later of the pruning but there is no need to run the propagator again. When using adjacency lists, the two relevant lists are updated immediately when the pruning occurs. When the constraint is notified of a pruning, it tests whether the lists need to be updated. If not, the constraint is not queued for propagation. Hence when using adjacency lists the propagator does some limited fixpoint reasoning.

3.5.2. Priority queueing and triggering

EGCC places triggers on the upper and lower bounds of all cardinality variables. If it is using incremental graph maintenance, it is notified individually of each value that is removed from a target variable. Otherwise, it is notified of changes to target variables, specifying the variable affected but not the value(s) removed.

The EGCC is triggered in one of three ways depending on configuration:

- *Normal priority:* The propagator is executed whenever it is notified of any event.
- *Low priority:* The propagator is *queued* (added to the constraint queue if not already present) for any event.
- *Low priority with incremental graph:* The propagator is queued for any event from a cardinality variable. For the target variables, the propagator is queued whenever it is notified of a value removal that is not already reflected in the adjacency lists.

3.5.3. Important Edges and Dynamic Triggers

The edges of the residual graph can be partitioned into important and unimportant (as discussed in §3.3.8). Only the removal of an important edge can cause pruning of the target variables. The number of calls to Régin's algorithm can be reduced by ignoring the removal of unimportant edges.

The algorithm presented by Gent et al [6] records the edges T that Tarjan's algorithm uses in its internal proof that each SCC is strongly connected. Tarjan's algorithm performs a depth-first search (DFS) in the residual graph R. The edges of R which are traversed by the DFS are included in T. The

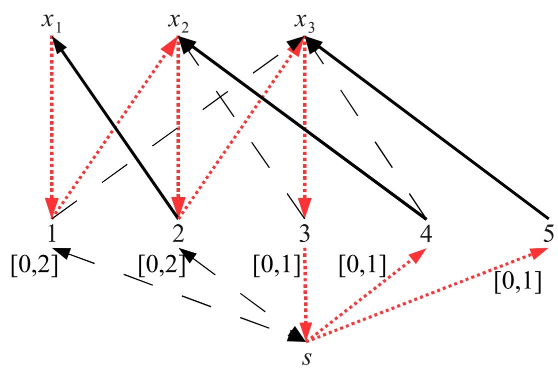


Figure 6: Finding important edges (T) in $Res(N(K), f)$. The flow f uses edges $(x_1, 1)$, $(x_2, 2)$ and $(x_3, 3)$. The DFS tree of Tarjan's algorithm is shown in red dotted thick lines. The three other edges in T are shown in thick black lines. Three edges corresponding to domain values are unimportant.

algorithm also maintains an integer named lowLink for each vertex. During the DFS, the lowLink values are 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 R are not included in T. This algorithm is also correct for EGCC; figure 6 shows an example of finding T for an EGCC constraint.

While the edges in T remain in the residual graph, each component will remain strongly connected and therefore no pruning is possible. This method yields at most $2r + d$ edges that correspond to domain values. Compared to Katriel's theoretical bound of $3r$ [12], there are $d - r$ spurious edges. However the method is simple and fast, with only minimal instrumentation of Tarjan's algorithm and no change to the time bound.

Two variants of AllDifferent were implemented by Gent et al [6] based on important edges. The first variant used dynamic triggers (movable triggers that are restored on backtracking), moving at most $2r + d$ value triggers each time Tarjan's algorithm is executed. Dynamic triggers are substantially more expensive than static triggers, and in experiments the cost of dynamic triggers outweighed the benefit in most cases.

The second variant records the important domain values in backtracking arrays. When the propagator is triggered, it returns immediately if no important value has been removed. The main cost is backtracking the arrays by block copying. In experiments this was a minor improvement with an average 6% speed-up. This approach is referred to as *internal dynamic triggers* because it simulates dynamic triggers within the constraint.

The internal dynamic triggers method of Gent et al [6] can be trivially adapted for EGCC. The algorithm for constructing set T is used unchanged. For each variable, a list of values is stored corresponding to edges in T. The lists are linked lists, stored in a block of backtracking memory with size $O(3r + d)$. This allows $O(1)$ append, quick iteration, and linear-time clear.

The propagator is changed in only two places. Tarjan's algorithm is changed to record the T values into the backtracking array. When a cell with target variables X_{cell} is triggered, each changed variable $x_i \in X_{cell}$ is checked against its list of T values. If no T values have been deleted, then the target variables are not pruned for that cell[6]. Cardinality variables are pruned regardless. This approach is evaluated in §5.7.

4. Pruning the cardinality variables

In this section I describe three algorithms for pruning the cardinality variables. The first is a simple approach that counts values in the target domains, it does not make use of the flow network. The second approach is to use the simple algorithm and add an implied sum constraint. The third approach computes a maximum and minimum flow for a particular value. In all cases, the algorithm described is run after pruning the target variables.

These three methods are compared empirically in §5.8.

4.1. A simple algorithm

For a domain value a, a simple upper bound is the number of target variables that have a in their domain. A lower bound is the number of target variables that are assigned to a.

For each value a, when not using incremental graph maintenance, the algorithm iterates through all target variables and computes the upper and lower bound. When using incremental graph maintenance, the upper bound is already known: it is the length of the adjacency list for a. The algorithm finds the lower bound by iterating through the adjacency list of a and counting assigned variables.

If dynamic partitioning is used, the algorithm only processes values in those cells of the constraint that have been triggered. Assigned variables are removed from active cells of the constraint, therefore values that only occur in assigned variables will not be processed.

The algorithm described above is stateless (i.e. requires no backtracking state) and quadratic ($O(rd)$), and it behaves well combined with dynamic partitioning and incremental graph maintenance. A stateless $O(r + d)$ algorithm is possible but preliminary experiments did not show any benefit, and this algorithm does not partition dynamically, therefore I disregarded it. It is also possible to construct a stateful $O(d)$ algorithm, by maintaining the number of variables assigned to each value in a set of backtracking integers. I avoided this because it requires backtracking memory.

[6]If dynamic partitioning is not used, consider the constraint to have one cell containing all target variables.

4.2. An implied sum constraint

A second approach is to use the simple algorithm and add an implied sum constraint over the cardinality variables. The total occurrences of all values must equal the number of target variables.

$$\text{egcc}(X, V, C) \ \wedge \ \sum C = r$$

This implied constraint is sound iff all values in the domains of target variables are in V, and therefore have a corresponding cardinality variable.

This is the approach used by Gecode [27][7]. However, in Gecode the definition of EGCC is slightly different: the variables in X are only allowed to take values in V. Therefore the sum constraint is always sound in Gecode.

4.3. A flow network algorithm

Quimper et al [18] proposed an algorithm based on the flow network $N(K)$. For a value a and cardinality variable c_a, the algorithm finds a maximum flow containing the minimum occurrences of a. This is used to prune the lower bound of c_a. Similarly, it finds a maximum flow containing the maximum occurrences of a to prune the upper bound of c_a.

This is an expensive method, but it provides the maximum possible pruning under the assumption that domains of cardinality variables are an unbroken interval.

4.3.1. Pruning lower bounds

The algorithm described by Quimper et al is as follows. Take an existing maximum flow f that respects the upper bounds for all values. Remove all units of flow that pass through value-vertex a, to form the reduced flow f_a. Similarly, remove vertex a and all incident edges from $N(K)$ to form a new network $N(K)_a$. Using the Ford-Fulkerson algorithm on network $N(K)_a$, augment f_a to find a maximum flow f'_a from s to t.

f'_a represents a maximum assignment to the target variables X such that a is not used and all values are within their upper cardinality bounds. Therefore to complete the assignment to X, there must be $r - |f'_a|$ occurrences of a, therefore $c_a \geq r - |f'_a|$.

The implementation makes use of the transpose graph and is almost identical to that described in §3.4.1, with three changes: the algorithm does not stop when it encounters a variable-vertex with no augmenting path; the graph $N(K)_a^T$ is used in place of $N(K)^T$; and the algorithm halts if the size of the flow reaches $r - \underline{c_a}$, because in this case it is not possible to prune c_a.

The time required to prune all cardinality lower bounds is $O(r^2 d)$ [18], or $O(r\delta)$ where δ is the number of edges in $N(K)$. This is because the algorithm seeks at most r augmenting paths.

[7]Guido Tack, personal communication

4.3.2. Pruning upper bounds

To find a new upper bound for value a, the flow through edge (s, a) in $N(K)$ is maximized while observing the lower cardinality bound for all other values. Quimper et al prove that this can be done in $O(r^{2.66})$ time. To find the upper bound of a, they start with a non-maximal flow f_l with exactly $\underline{c_b}$ occurrences of each value b (therefore the maximum number of free variables). The goal is to maximize the flow using paths from a to a free variable. If a value-vertex is not used in f_l then it cannot be part of a path (excluding s) from a to a free variable in $Res(N(K), f_l)$. The number of reachable vertices is no more than $2r + 1$. The authors identify the network as a special case and cite a proof that the maximum flow can be found in $O(r^{2.66})$ time.

The implementation used here is simpler. It begins with a maximum flow f that respects the lower bounds for all values. To find the upper bound for a, the algorithm maximizes the flow through (s, a) by finding augmenting paths with a BFS in $N(K)$ (excluding s) starting at a and ending at a value-vertex b where $f(s, b) > \underline{c_b}$ (i.e. the flow through b is greater than its lower bound). The path is applied to increase the flow through a and decrease b. The algorithm halts if the size of the flow through a reaches $\overline{c_a}$, because in this case it is not possible to prune c_a. The final flow through edge (s, a) is the new upper bound for c_a.

In common with Quimper's algorithm, the number of reachable vertices is $2r + 1$ or fewer. To prune all capacity variables C, the time bound is $O(|C|r^3)$, a factor of $r^{1/3}$ less efficient.

5. Experimental Evaluation

In this section I describe the context of the experimental evaluation. Then I present two groups of experiments. First I compare various algorithms and optimizations for pruning the target variables, in experiments one to five. Secondly, I compare algorithms for pruning the cardinality variables in experiment six. Finally, in §5.10, the best propagation method for the target variables is compared against a careful but unoptimized implementation of Régin's algorithm. Also, the best EGCC propagator is compared against the decomposition of EGCC into a set of occurrence constraints, demonstrating the utility of EGCC as a global constraint.

5.1. Benchmark Set

In this section I describe the problem classes and instances used to compare propagation algorithms.

5.1.1. Car Sequencing

The car sequencing problem [9] (prob001) is to sequence cars on a conveyor through a factory. There are a number of optional parts that may be fitted to the cars, and each optional part has a corresponding machine which fits the part. For an option i, the machine cannot accept more than p_i cars in every q_i.

Therefore, in every contiguous subsequence of length q_i there must be no more than p_i cars requiring the option.

There are a number l of different types of car, where each type has a set of options that it requires. A fixed number of each type is required in the sequence. Finally, the length n of the sequence is given.

Three models for this problem are presented below. They all share a common core. There is an explicit representation *seq* of the sequence. This is an array of length n of variables with domain $\{0 \ldots l-1\}$ (representing the type of car). An EGCC constraint is placed on *seq*, to enforce the required number of each type of car.

All models also contain a two-dimensional array *optused* of Boolean variables. For each sequence index j and option k, $\mathit{optused}[j,k]$ indicates whether the car at position j requires option k. Each element $\mathit{optused}[j,k]$ is connected to $\mathit{seq}[j]$ by a binary table constraint.

Model A

Régin and Puget presented an encoding of the capacity constraints for the machines as a set of EGCC constraints [25]. It is a very complex model and I do not reproduce it in full.

For each option i, there are $n + q_i - 1$ subsequences of *seq* to consider. For each subsequence, we need to state that no more than p_i of the cars require option i. This is done using $n \times q_i$ extra variables, and q_i EGCC constraints (each with n target variables).

There is also a cardinality variable for each subsequence, giving the number of cars in the subsequence that do not require option i. Two consecutive subsequences of the same length overlap by $q_i - 1$ cars, therefore the corresponding cardinality variables cannot differ by more than 1, and the difference is easily determined. A set of logic and arithmetic constraints are added to capture this fact.

The key advantage of Régin and Puget's model is that the EGCC constraints combine subsequence capacities with constraints on the whole sequence. Each car type that requires option i is represented in the auxiliary variables, and the required number of that car type is enforced by all the EGCC constraints.

Régin and Puget do not report the level of consistency that was used for the EGCC cardinality variables. However, on instance 2, Régin and Puget report 9355 fails in ILOG Solver [25]. Minion performs 9452 left branches (using the simple cardinality pruning algorithm in §4.1). Instance 1 is also similar (0 fails in ILOG Solver, 113 left branches in Minion). This suggests that the model, propagation and variable ordering may be equivalent.

Model B

In car sequencing, a single *sequence* constraint [33] may be used to represent the capacity constraints for one option. The sequence constraint for option i is given parameters p_i and q_i, and is posted on variables $\mathit{optused}[i,*]$. It restricts the number and position of occurrences of value 1 in $\mathit{optused}[i,*]$.

Van Hoeve et al [33] proposed an encoding of the sequence constraint as a regular constraint (i.e. a constraint that recognizes a regular language). In this case the language is the set of all assignments to $optused[i, *]$ that satisfy the capacity constraints for option i. The corresponding deterministic finite automaton (DFA) has $O(2^{q_i})$ states. A cost parameter is added to fix the total number of cars with option i (i.e. to fix the number of 1's in $optused[i, *]$), and the cost-regular propagator [3] is used to enforce GAC on the resulting constraint.

Minion does not have cost-regular or regular propagators, therefore the constraint is encoded into table constraints. First, the DFA is augmented with a counter that counts the number of 1's in the sequence. Only sequences with the correct number of 1's are accepted by the augmented DFA. The number of states is increased to $O(n2^{q_i})$. The augmented DFA is encoded into a set of ternary table constraints as described by Quimper and Walsh [21], and GAC is enforced on these table constraints. This is equivalent to enforcing GAC on the original cost-regular constraint.

Model AB

Model AB is the combination of models A and B. This is very similar to model '(C) A + REG with cost' in van Hoeve et al [33].

Variable and value ordering

All three models use the variable and value ordering by Régin and Puget [25]. First, the options are ordered according to a measure of how tightly constrained they are. For each option i, the measure uses the *demand* of i, denoted k_i, which is the number of cars in the sequence that require it. The *slack* of option i is $n - q_i(k_i/p_i)$ (where low slack indicates the option is tightly constrained[8]).

The *optused* variables are searched. First, the options are ordered by slack, with the least slack first. For each option, the variables are branched from the middle out (i.e. at each step the unassigned variable closest to the middle of the sequence is selected). Finally, the value order is 1 then 0.

80 instances were used. Instances 0 to 4 are from Régin and Puget[9], and 5 to 79 are the other instances given on CSPLib (prob001) [9].

5.1.2. Magic Sequence

The magic sequence problem [9] (prob019) is to find a sequence of length n such that element i in the sequence is the number of occurrences of i in the sequence. It is modelled as a list X of n variables with domain $\{0 \dots n-1\}$.

[8]Régin and Puget claim that negative slack means the capacity constraint for the option cannot be satisfied [25]. This is not true because the ends of the sequence are a special case. Consider a problem where $n = 8$ and an option i has parameters $p_i = 2$, $q_i = 3$ and demand $k_i = 6$. The slack for option i is -1, and the capacity constraint can be satisfied with the *optused* sequence $\langle 1, 1, 0, 1, 1, 0, 1, 1\rangle$.

[9]For instances 0 and 3 the option ordering derived from slack is not the same as that reported by Régin and Puget [25], which may have been adjusted by hand.

c_1	0	0	1	1	2	2
c_2	0	1	0	2	1	2
c_3	0	1	2	0	2	1
c_4	0	2	1	2	0	1
c_5	0	2	2	1	1	0

Table 1: EFPA example with $v = 5$, $q = 3$, $\lambda = 2$, $d = 4$

There is one constraint: EGCC$(X, \langle 0 \dots n-1\rangle, X)$. The variables are searched in index order, and values are explored in ascending order. Instances were generated for $n \in \{20, 30, 40, 50, 100, 150, 200, 300\}$.

5.1.3. Equidistant Frequency Permutation Arrays (EFPAs)

The EFPA problem [11] is to find a set (often of maximal size) of codewords, such that any pair of codewords are Hamming distance d apart. Each codeword (which may be considered as a sequence) is made up of symbols from the alphabet $\{1, \dots, q\}$, with each symbol occurring a fixed number λ of times per codeword. A fourth parameter v is the number of codewords in the set. Typically v would be maximized. Table 1 shows an example of an EFPA.

The problem is modelled as a two-dimensional array of variables where each row represents a codeword. The model is given by Huczynska et al [11] (the non-Boolean model with the implied constraint set). The variable and value ordering described there is used. The 24 instances of EFPA used in the experiments in [11] are also used here, with two added: $d = 4$, $q = \lambda = 5$, $v \in \{11, 12\}$. This provides a mixture of 13 satisfiable instances and 13 unsatisfiable or unknown instances.

Each row of the model has an EGCC constraint with $q\lambda$ target variables to enforce λ occurrences of each symbol. There are also EGCC constraints with λ target variables used in the implied constraints.

5.1.4. Round-robin Sports Scheduling

The round-robin sports scheduling problem [9] (prob026) is to schedule games among n teams on $n/2$ pitches over $n-1$ weeks. The model and variable and value ordering is given in [6], and EGCC constraints are added to enforce the requirement that each team plays on each pitch at most twice. This gives $n/2$ EGCCs with $2(n-1)$ target variables and capacities $0 \dots 2$ for all values. Instances were generated with $n \in \{10, 12, 14, 16\}$.

5.2. Experimental Setup

For all experiments I use Minion as described in §2.5[10]. The instances are not preprocessed. I used a timeout of 1800s. The experiments were run on a Linux (Ubuntu 9.10) server with two Intel Xeon quad-core E5520 CPUs clocked at 2.27GHz and 12GB of RAM.

[10]Source code for the solver is available at `http://www.cs.st-andrews.ac.uk/~pn/egcc/`

Minion performs a binary search where the left branch assigns a variable. It counts left branches. The speed of search is measured by dividing the number of left branches by time taken, this is referred to as *branch rate*.

In this setup, timings (and branch rates) exhibit some variation. To measure the variation, I used the PriorityQ-IncMatch-IncGraph propagator (§ 5.5), and measured the branch rate twice for each benchmark (with a timeout of 1800s). For each instance, the absolute difference between the two rates was divided by the smaller rate to obtain a proportional difference. The mean of these values is 0.03 (i.e. the larger branch rate is 3% larger than the smaller one) and the maximum is 0.50. Those instances that completed in less than 0.1s showed the most variation; excluding those, the mean is 0.03 and maximum is 0.17.

In all experiments below, the median of three runs is used. For each experiment comparing algorithms A and B, to determine statistical significance I used the Wilcoxon paired signed-rank test implemented in R [22]. The branch rates of A and B are measured for all benchmark instances. The null hypothesis is that the branch rates are drawn from the same distribution (i.e. A and B run at the same speed). The difference between A and B is deemed to be significant if the probability of the null hypothesis is less than 0.01.

The various implementations of EGCC were extensively tested and de-bugged, and all variants report the same branch count on instances that complete within the time-out.

5.2.1. Solver architecture

The EGCC propagators make use of some Minion features that may not be available in all solvers. Perhaps the most important is notification of which variables have been changed, and (when using the incremental graph) which values have been pruned. This granularity of events is widely available however (for example in Gecode via advisors [15] and Choco [14]).

Another important consideration is memory architecture. Minion allows propagators to have both backtracked and non-backtracked memory. The backtracked memory for all propagators is blocked together, and cannot be allocated or freed during search. Memory is backtracked by copying the block, which is very efficient when the amount of backtracked memory is small and static. Memory architecture affects most of the optimizations for EGCC, therefore experimental results that are marginal could be reversed with a different architecture. Different data structures may be required with a different architecture.

In contrast to Minion, Gecode backtracks all state, and it does so by copying. When search branches Gecode traverses a tree of objects (including constraints and variables) and copies each individually. This architecture has very different properties to Minion. Entailment (§3.3.9) is a case in point: in Gecode, if a constraint is entailed, it is removed from the tree (and its triggers are removed) thus the cost of copying the constraint and its triggers is removed. In Minion triggers are not copied when search branches, and any backtracking state the constraint has cannot be de-allocated. Therefore the potential gain from detecting entailment is much less in Minion. Gecode employs entailment for

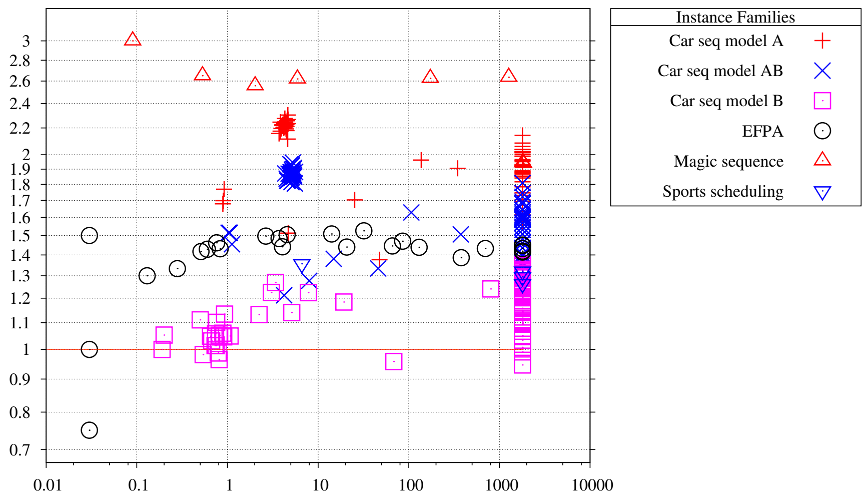


Figure 7: Speedup of Baseline-Régin compared to Baseline-Quimper. The graph is a scatter-plot, with each point comparing results on a single instance. The x-axis represents the run time of Quimper's algorithm to solve the instance. The y-axis gives the speedup obtained by using Régin's algorithm instead of Quimper's algorithm. A ratio of 1 indicates that the two methods run at the same speed, with ratios higher than 1 indicating that Régin's algorithm is faster, and ratios less than 1 indicating that Quimper's algorithm is faster. The ratio is calculated by dividing the branch rate with Régin's algorithm by that with Quimper's algorithm. In this graph we can see that Régin's algorithm almost always performs substantially better than Quimper's algorithm.
All subsequent graphs labelled 'Speedup of X compared to Y' follow the same conventions, where in this case X=Baseline-Régin and Y=Baseline-Quimper.

GCC[11].

5.3. Experiment One: Comparing Quimper's and Régin's algorithms

In this section I compare the two basic algorithms for pruning the target variables. To do this, various other choices must be made. These choices are mainly based on the current state-of-the-art from the literature. I use a priority queue where EGCC has a low priority (§3.5.2), incremental matching (§3.3.1), and incremental graph maintenance (§3.5.1), including fixpoint reasoning. I do not use dynamic partitioning (§3.4.2), assigned variable removal (§3.3.6), dynamic triggers (§3.5.3), or the transpose graph (§3.4.1). The weakest algorithm is used to prune cardinality variables (§4.1). The two algorithms are referred to as Baseline-Régin and Baseline-Quimper.

Figure 7 shows the experimental results comparing Régin's algorithm to Quimper's. The results are strongly in favour of Régin's algorithm, despite the better worst-case bound of Quimper's algorithm. The results are statistically

[11]Guido Tack, personal communication

Instance	Search nodes	Calls	Proportion in flow algorithm	Proportion in Tarjan's algorithm
EFPA-4-4-4-8	27100	86824	14%	71%
Magic sequence 40	145	40157	1.4%	67%
Car seq A instance 1	113	6400	5.7%	65%
Car seq B instance 1	111	116	7.3%	92%
Car seq AB instance 1	111	6064	5.8%	64%
Sports scheduling 10	36926	324860	7.4%	76%

Table 2: Instructions spent in the flow algorithm and in Tarjan's algorithm, as a proportion of the propagator Baseline-Régin. To avoid inlining, the solver was re-compiled without optimizations.

significant, with a mean speed-up of 1.62 times. Recall that the speed of the whole solver is measured, so 1.62 is a lower bound on the true speed-up of the EGCC propagator.

The performance of the two algorithms is closest on car sequencing model B. These instances contain only one EGCC constraint and a large set of table constraints (typically over 1000) and other constraints. Therefore the potential to speed them up by improving the EGCC algorithm is limited.

Quimper's algorithm intends to speed up the first stage of the process — computing a maximum flow or matching — by using a more sophisticated algorithm. The second stage is almost identical to Régin's, except that it must be performed twice in Quimper's algorithm. To investigate further, I profiled Baseline-Régin using Callgrind [35]. Table 2 shows the proportion of CPU instructions spent in the flow algorithm and Tarjan's algorithm. The solver was profiled on one easy satisfiable instance from each problem class. For all six instances, the algorithm spends over 60% of its CPU instructions in Tarjan's algorithm, and less than 15% in the flow algorithm. This is consistent with the empirical results: if Tarjan's algorithm is the more expensive stage, it would be counterproductive to run it twice in order to speed up the first stage.

In all six cases, the proportion of instructions spent in the maximum flow algorithm is surprisingly low, with the bulk of instructions spent in Tarjan's algorithm. While the flow algorithm has a worse upper bound, Tarjan's algorithm always reaches its upper bound (§3.1.5).

Based on these results, only Régin's algorithm is used for the rest of the experiments.

5.4. Experiment Two: Making use of the transpose graph

In §3.4.1 I described a change to Régin's algorithm intended to speed up the computation of a maximum flow. To evaluate this I use the same experimental set-up as in the previous experiment, and simply compare Régin's original algorithm with the variant. Figure 8 is a plot of the results. It appears that measurement noise hides the difference between the two algorithms. The difference is not statistically significant.

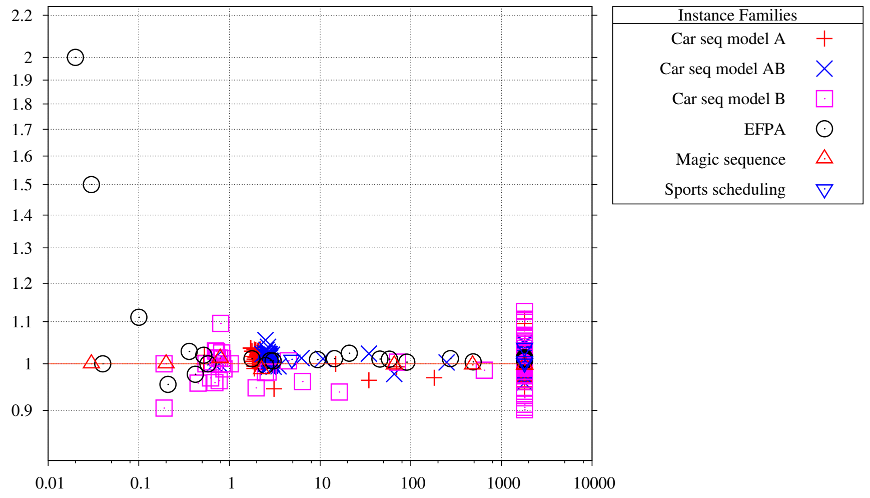


Figure 8: Speedup of Transpose compared to Régin's original algorithm

Instance	Search nodes	Calls	Standard Régin's algorithm	Transpose	Instructions in max flow for standard
EFPA-4-4-4-8	27100	86824	267m	247m	14%
Magic seq 40	145	40157	315m	297m	1.4%
Car seq A 1	113	6400	290m	193m	5.7%
Car seq B 1	111	116	4.63m	4.62m	7.3%
Car seq AB 1	111	6064	284m	188m	5.8%
Sports sched 10	36926	324860	1724m	1201m	7.4%

Table 3: Instruction counts for finding or repairing a maximum flow, with and without the transpose graph. To avoid inlining, the solver was re-compiled without optimizations.

To obtain more exact data, I profiled the solver using Callgrind [35]. The profiler provides the total number of instructions spent in a function and other functions it called. Table 3 shows instruction counts for finding or repairing the maximum flow with and without the transpose graph. The proportion compared to the whole propagator is also given.

Using the transpose graph does give substantial gains in some cases. For example on the car sequencing A instance, it is 33% better. However, on that instance, the overall gain is very low because the propagator only spends 5.7% of its instructions in the maximum flow algorithm.

Based on the results of this experiment, from here on I use only Régin's algorithm with the transpose graph.

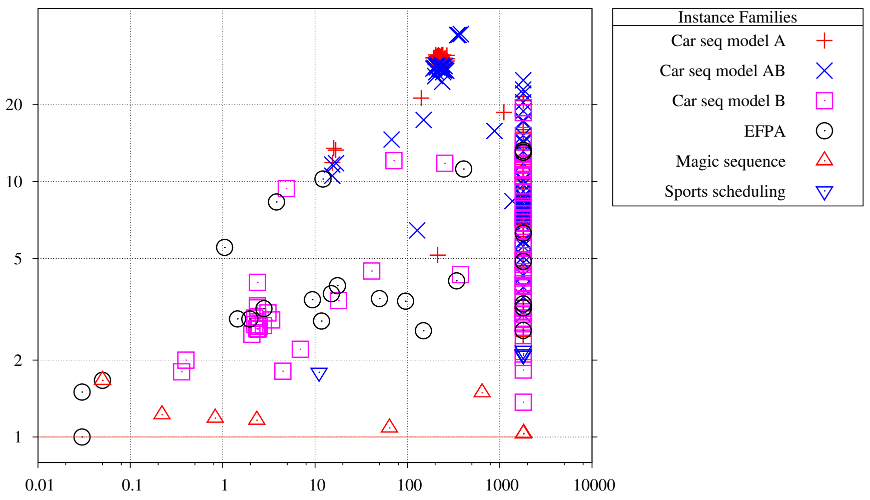


Figure 9: Speedup of PriorityQ compared to Simple

5.5. *Experiment Three: Standard optimizations of Régin's algorithm*

In this section I experiment with optimizations found in the literature, and investigate whether they are worthwhile. The following variants of Régin's algorithm are used.

Simple The algorithm described in §3.1 with the transpose graph optimization (§3.4.1), run at normal priority. The weakest algorithm is used to prune cardinality variables (§4.1).

PriorityQ The Simple algorithm run at low priority as described in §3.5.2.

PriorityQ-IncMatch PriorityQ plus incremental matching as described in §3.3.1.

PriorityQ-IncMatch-IncGraph PriorityQ-IncMatch plus incremental graph maintenance as described in §3.5.1.

Figure 9 shows that it is worthwhile to use the priority queue. All instances are faster with PriorityQ compared to Simple. Even Magic Sequence instances (with one constraint) benefit from PriorityQ because the propagator is called once for multiple variable events.

Figure 10 shows that it is worthwhile to use incremental matching. Almost all instances are faster with PriorityQ-Incmatch compared to PriorityQ, with very substantial speedups in some cases.

Finally, Figure 11 shows that in most cases it is worthwhile to use PriorityQ-IncMatch-IncGraph compared to PriorityQ-IncMatch. The main exception is the magic sequence problem, where all eight instances are slower with PriorityQ-IncMatch-IncGraph.

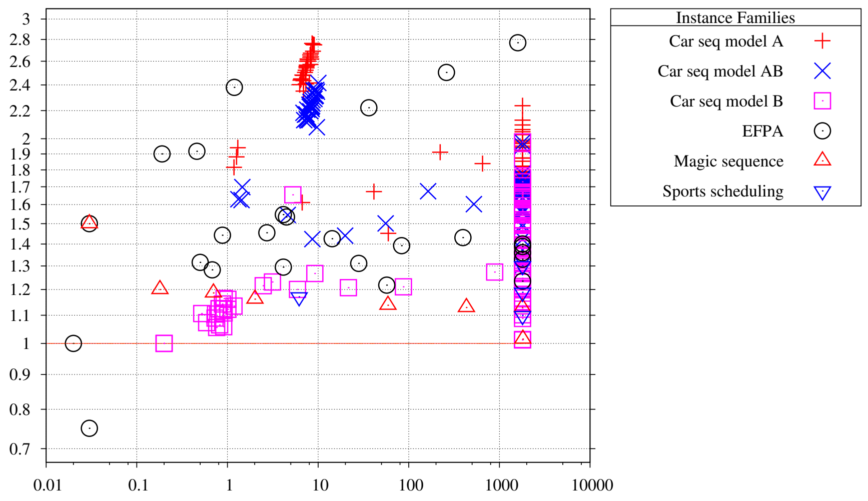


Figure 10: Speedup of PriorityQ-IncMatch compared to PriorityQ

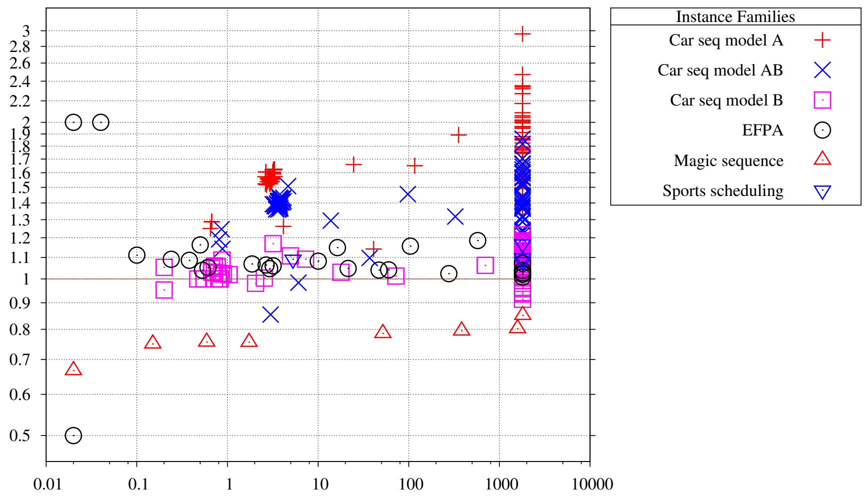


Figure 11: Speedup of PriorityQ-IncMatch-IncGraph compared to PriorityQ-IncMatch

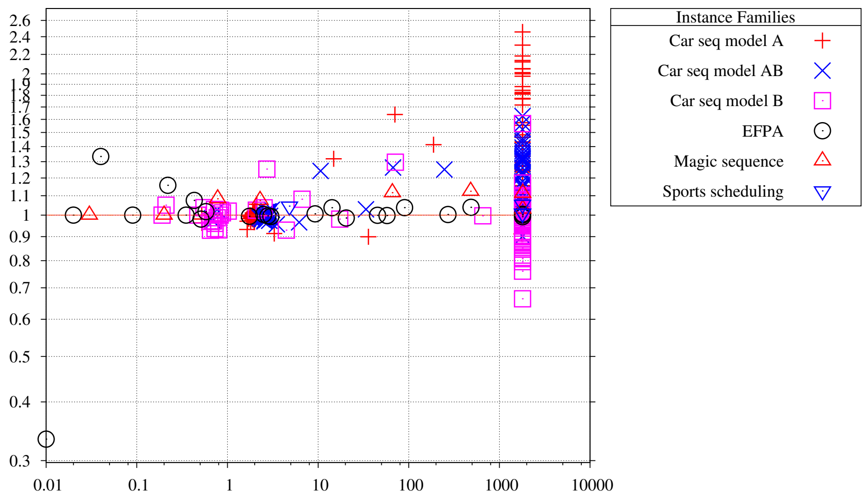


Figure 12: Speedup of Baseline-AVR compared to Baseline

For all three comparisons, the difference is statistically significant. Based on these results, I use PriorityQ-IncMatch-IncGraph as a baseline for all subsequent experiments.

The results for PriorityQ and IncMatch are broadly similar for AllDifferent [6]. However, for AllDifferent the speedup for IncGraph is less substantial [16]. Also, when using dynamic partitioning, IncGraph is detrimental for most instances [16].

5.6. Experiment Four: Assigned Variable Removal and Dynamic Partitioning

In this experiment I evaluate assigned variable removal (AVR) and dynamic partitioning. These two optimizations are closely related: dynamic partitioning subsumes AVR, because it partitions assigned variables into a singleton cell. I compare the following three variants experimentally.

Baseline The same as PriorityQ-IncMatch-IncGraph in the previous section

Baseline-AVR Baseline with assigned variable removal (§3.3.6)

Baseline-Cell Baseline with dynamic partitioning as described in §3.4.2.

The second stage of Régin's algorithm (i.e. Tarjan's algorithm) is often more expensive than the first stage (Table 2). As discussed in §3.4.2, dynamic partitioning improves the time bound of Tarjan's algorithm from $\Theta(\delta)$ to $O(\delta)$. AVR does not have this effect.

Figure 12 shows that it is worthwhile to remove assigned variables in most cases, and for some of the most difficult instances. In the best case, it sped up the solver by 2.5 times, and in the worst case slowed it down by about 35%.

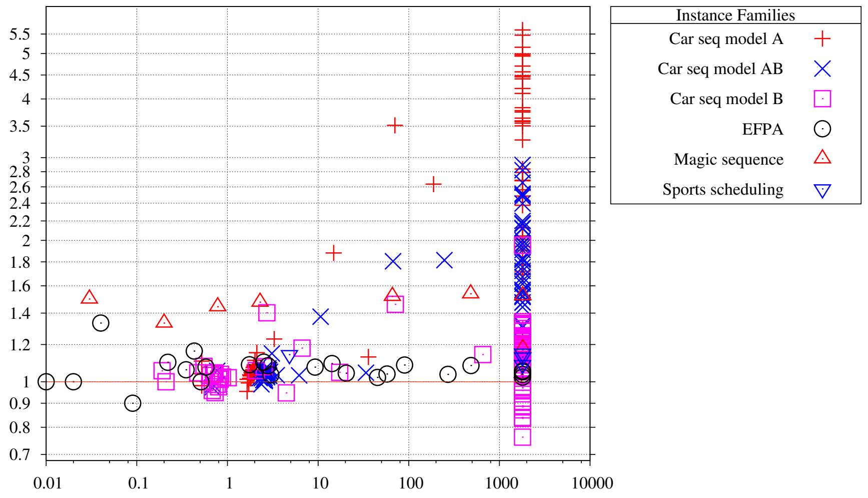


Figure 13: Speedup of Baseline-Cell compared to Baseline

The comparison of Baseline-Cell against Baseline is plotted in Figure 13. In this case the results are much more pronounced than AVR, with a speed up of 5.6 times in the best case. Car sequencing models A and AB and the magic sequence problem benefit substantially from dynamic partitioning. EFPA and sports scheduling show a less substantial benefit. Car sequencing model B also shows benefit, with 67/80 instances running faster with dynamic partitioning, even though there is only one EGCC constraint. The mean average speedup is 1.56.

For both comparisons, the difference is statistically significant.

For AllDifferent, dynamic partitioning is very effective [6], yielding a mean speedup of 2.98 times (with the assignment optimization). Dynamic partitioning superficially appears to be more effective for AllDifferent, however the two benchmark sets are entirely different.

5.7. Experiment Five: Internal Dynamic Triggers

All previous optimizations were intended to speed up some part of the propagator. In contrast, internal dynamic triggers (IDT, §3.5.3) is intended to reduce the number of times that Régin's algorithm is called.

Dynamic partitioning reduces the cost of pruning target variables, therefore it reduces the potential for internal dynamic triggers to save time. Therefore, I evaluate internal dynamic triggers both with and without dynamic partitioning. Four variants are used.

Baseline The same as Baseline in the previous section

Baseline-IDT Baseline with internal dynamic triggers (§3.5.3)

Instance	Search nodes	Calls with Baseline	Calls with Baseline-IDT
EFPA-4-4-4-8	27100	86824	57260
Magic sequence 40	145	40157	8116
Car seq A instance 1	113	6400	1856
Car seq B instance 1	111	116	114
Car seq AB instance 1	111	6064	1662
Sports scheduling 10	36926	324860	196472

Table 4: Calls to Régin's algorithm comparing Baseline to Baseline-IDT

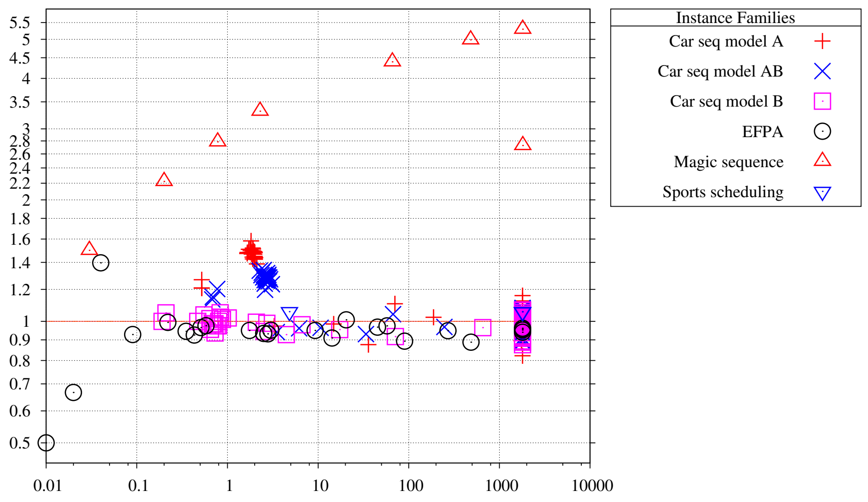


Figure 14: Speedup of Baseline-IDT compared to Baseline

Baseline-Cell Baseline with dynamic partitioning.

Baseline-Cell-IDT Baseline-Cell with internal dynamic triggers.

Table 4 shows the number of calls to Régin's algorithm with Baseline and Baseline-IDT for the six easy problems used previously. It shows that the dynamic triggers approach can substantially reduce the number of calls. The most encouraging is magic sequence 40 where the number of calls is reduced by 80%.

Figure 14 shows the empirical results comparing Baseline-IDT to Baseline. The magic sequence problem benefits the most from IDT, but this could be a red herring because of its very unusual structure. For other problems, the difference ranges from 0.5 to 1.6 times faster. Overall the mean speedup is 1.15. This indicates that the overhead of maintaining and backtracking the internal dynamic triggers cancels out the benefit in most cases. Although the two algorithms are similar, the difference is statistically significant.

As expected, dynamic partitioning reduces the benefit of dynamic triggers. Baseline-Cell-IDT is 7% slower on average than Baseline-Cell on the benchmarks. Baseline-Cell-IDT was faster for 69 of 278 instances. The maximum speed-up was just 7%. The difference is statistically significant.

As discussed in §3.5.3, the cost of collecting the trigger values is negligible, so it seemed likely that IDT would help, particularly with long constraints. However, this is not what was observed. Dynamic triggers were also unsuccessful in AllDifferent when applied with dynamic partitioning [6].

5.8. Experiment Six: Pruning the Cardinality Variables

In this experiment I compare the three methods of pruning the cardinality variables described in §4. Dynamic partitioning (Baseline-Cell) has been found to be a substantial improvement over Baseline, therefore I use Baseline-Cell and combine it with the three methods as follows.

Baseline-Cell The same as Baseline-Cell in the previous section. This employs the simple cardinality algorithm described in §4.1.

Baseline-Cell-Sum Baseline-Cell with the additional sum constraint as described in §4.2. For all benchmarks, the sum constraint is correct.

Baseline-Cell-Flow Baseline-Cell with the flow cardinality algorithm described in §4.3.

The three variants perform different levels of propagation (and are ordered from least to most powerful). In this experiment I compare run times rather than branch rates. I also do not evaluate on those instances where the cardinalities are constants. This leaves car sequencing models A and AB, magic sequence, and EFPA. The Wilcoxon paired signed-rank test was applied to run time rather than branch rate, with the result that each pair of methods are significantly different.

Since Baseline-Cell-Sum is an improvement of Baseline-Cell, I will compare these first. Figure 15 shows that the usefulness of the sum constraint depends very much on problem class. On magic sequence, it is consistently very useful. It is also useful on the majority of car sequencing problems where neither variant timed out.

As shown in Table 5, Baseline-Cell-Sum is able to solve one additional instance (magic sequence 300) within the time limit. For 20 instances of 109 it reduced the number of search nodes. These 20 instances consist of all eight magic sequences, the three unsatisfiable EFPA instances where $d = 3$, and nine of car sequencing model A.

There are a number of car sequencing model AB instances where Baseline-Cell-Sum reached the same fixed point faster at the root node. However, only one of these instances is also faster during search.

In conclusion, adding the sum constraint is low-risk and is sometimes very helpful, and would be a good default choice in place of Baseline-Cell.

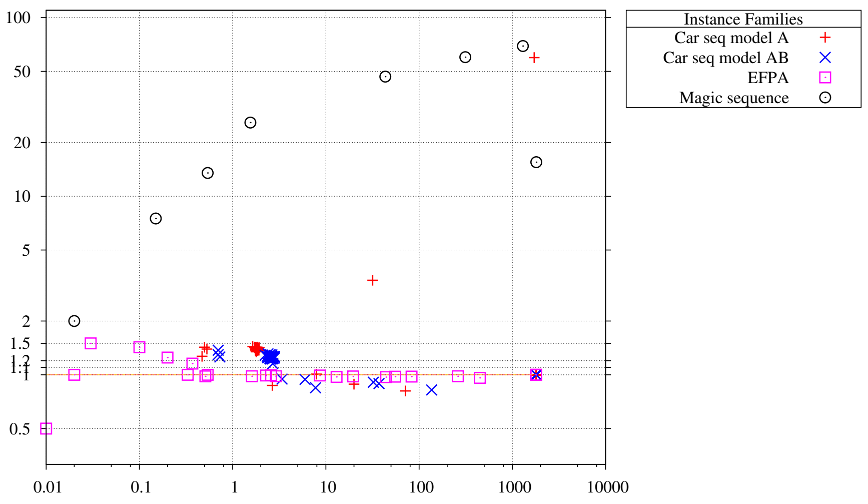


Figure 15: Time comparison between Baseline-Cell-Sum and Baseline-Cell. The x-axis is the time taken by Baseline-Cell and the y-axis is the proportion of total times, Baseline-Cell divided by Baseline-Cell-Sum. The timeout was 1800s.

	Instances solved (of 194)	Saved nodes vs Simple	Saved nodes vs Sum
Baseline-Cell	108	—	—
Baseline-Cell-Sum	109	20	—
Baseline-Cell-Flow	111	33	23

Table 5: For each cardinality algorithm: the number of instances solved; and the number of instances with a reduced node count vs the weaker algorithms.

Next Baseline-Cell-Flow is compared to Baseline-Cell-Sum. Table 5 shows that Baseline-Cell-Flow is more robust, solving two extra instances within the time limit (car sequencing 12 with models A and AB). Baseline-Cell-Flow explores fewer nodes for 21% of 111 benchmarks. The two instances that are solved only by Baseline-Cell-Flow appear in the upper right corner of Figure 16.

Baseline-Cell-Flow can be inefficient, as shown by Figure 16: in the worst case it is 48 times slower than Baseline-Cell-Sum on easy car sequencing instances. The bulk of this slow-down is at the root node: surprisingly the solver takes over 45s to reach the fixed point for most of the car sequencing benchmarks. In contrast Baseline-Cell-Sum never takes over 1.31s at the root node. For the first propagation of the EGCC, dynamic partitioning has no effect so all cardinality variables are pruned. For car sequencing this is very costly, but not for EFPA and magic sequence.

If the root node is excluded, in the worst case Baseline-Cell-Flow takes 5.73 times longer than Baseline-Cell-Sum (this would be 0.17 on the y-axis of Figure 16).

Both Sum and Flow are hugely more efficient than Baseline-Cell on the magic sequence problem. In both cases, this is mainly *not* because they explore fewer branches. Taking magic sequence 100 as an example, Baseline-Cell solves it in 385 branches, with 675192 executions of the EGCC propagator (average 1754 calls per branch). This is extremely pathological behaviour for an instance with only one constraint, and is caused by the cardinality variables being the same as the target variables. Baseline-Cell-Sum reduces this pathological behaviour, solving it in 242 branches and 41428 calls to the propagator (average 171 calls per branch). The speed-up of 47 times is much greater than the 1.59 times reduction in the number of branches, and greater than the 16 times reduction in the number of calls. Therefore adding the sum constraint reduces the average time taken per call to EGCC, as well as reducing the number of calls.

In conclusion, Baseline-Cell-Flow is risky, frequently slowing the solver down substantially and does not appear to be a good default choice. However, it is able to solve more instances within the half-hour time limit.

5.9. Experiment Seven: Comparing EGCC to AllDifferent

Given an efficient implementation of EGCC, is it worthwhile implementing GAC AllDifferent? The Baseline-Cell propagator was adapted by removing the cardinality variables and using $\{0, 1\}$ as the cardinality for all values. The adapted Baseline-Cell is compared to the SCC-AssignOpt variant of AllDifferent described by Gent et al [6], using the benchmarks from that paper. The difference is statistically significant, AllDifferent is 1.31 times faster on average.

5.10. Evaluating all optimizations combined

In the previous sections, I individually evaluated many efficiency measures for pruning the target variables of EGCC. In this section, I consider the effect of them all together. When using the simple cardinality algorithm, the most efficient variant is Baseline-Cell. Baseline-Cell is 51.5 times faster than Simple

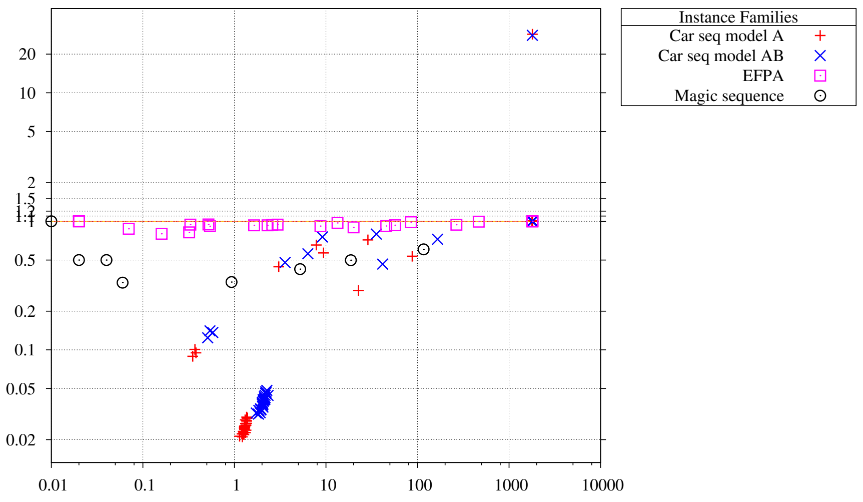


Figure 16: Time comparison between Baseline-Cell-Flow and Baseline-Cell-Sum. The x-axis is the time taken by Baseline-Cell-Sum and the y-axis is the proportion of total times, Sum divided by Flow. The timeout was 1800s.

on average, with a maximum speedup of 237 times. However, Simple does not include the priority queue optimization, which is ubiquitous and external to the propagator. Figure 17 compares Baseline-Cell with PriorityQ. The mean speed up is 4.11 times, and the maximum is 20.9. Baseline-Cell is a substantial improvement over PriorityQ, and this underlines the importance of implementing EGCC well.

Figure 18 is a plot of the nodes explored per second by Baseline-Cell. This gives an idea of the speed of the propagator on different classes of instances. The EFPA instances are very fast, exceeding 20,000 nodes per second in all cases, which is perhaps remarkable when maintaining GAC-On-X. In this case the constraints are quite short. For example on the instance $\langle 4, 4, 4, 9\rangle$ the longest EGCC has 16 target variables.

Magic sequence is by far the slowest class, with a single EGCC constraint whose arity is the length of the sequence. This class has extremely pathological behaviour with Baseline-Cell, making a very large number of calls to the propagator to reach a fixed point after each branch (e.g. on instance 100, average 1754 calls per branch). This is caused by the target and cardinality variables being the same, therefore the constraint triggers itself many times before reaching a fixed point.

Finally, I compare one of the best EGCC variants (Baseline-Cell-Sum) against the decomposition of the EGCC constraint into a set of occurrence constraints. The decomposition is as follows. For each constraint egcc(X, V, C), for each value $a \in V$ and corresponding cardinality variable $c_a \in C$, an occurrence constraint occurrence(X, a, c_a) is created, stating that c_a is the number of oc-

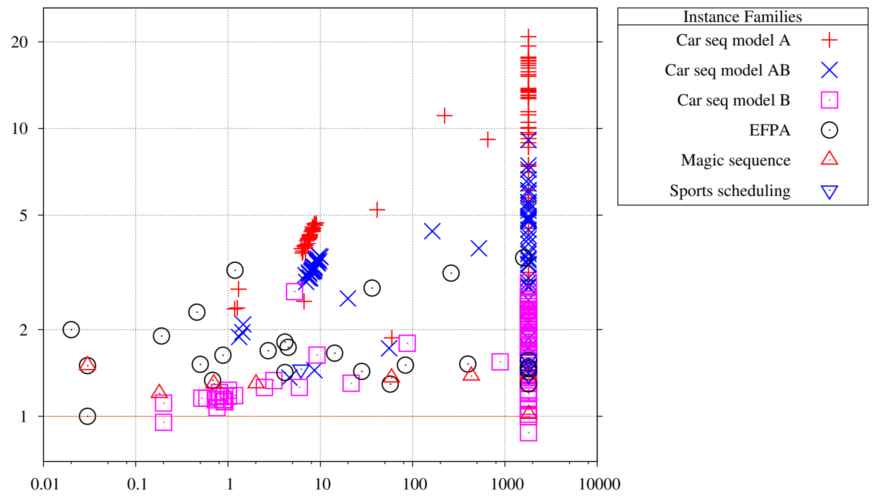


Figure 17: Speedup of Baseline-Cell compared to PriorityQ

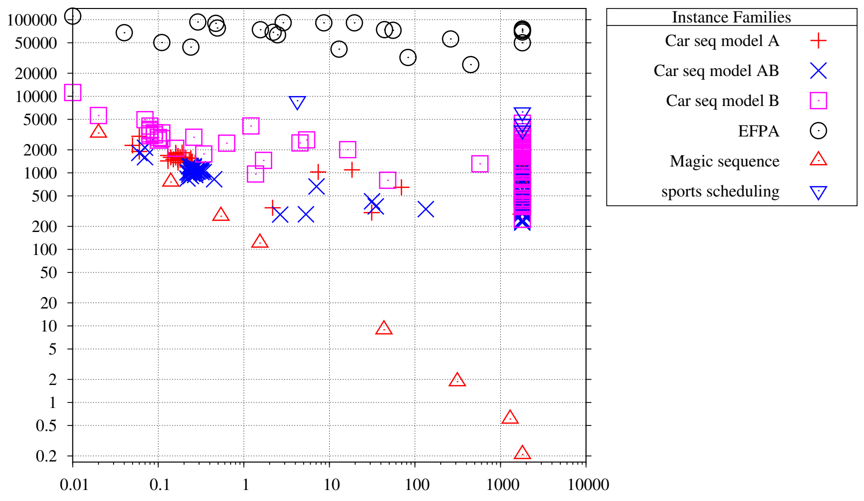


Figure 18: Plot of Baseline-Cell branches per second. The x-axis is Baseline-Cell runtime, and the y-axis the number of branches searched per second.

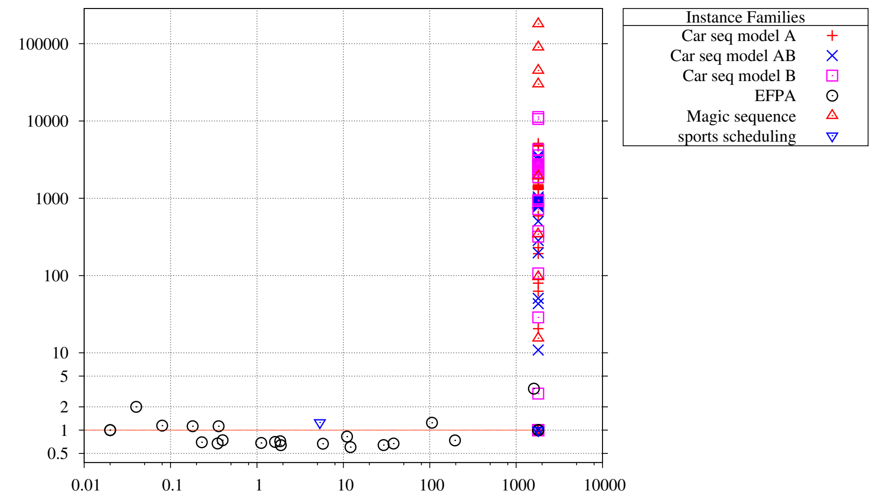


Figure 19: Time comparison between Baseline-Cell-Sum and Occurrence. The x-axis is the time taken by Occurrence, and the y-axis is the proportion of total times, Occurrence divided by Sum. The timeout was 1800s.

currences of a in X. The decomposition is referred to as Occurrence.

Figure 19 compares Baseline-Cell-Sum against Occurrence. Many instances were solved by Baseline-Cell-Sum and not by Occurrence, these are at the right-hand side of the plot. The speed-up can be many orders of magnitude, with the most extreme point being magic sequence 20, where Occurrence times out and Baseline-Cell-Sum takes 0.01s. Baseline-Cell-Sum appears to be faster for all problem classes except EFPA. Occurrence is faster for 15 instances (all from the class EFPA) and solves 21 in total, whereas Baseline-Cell-Sum solves 109. The proportion of run times ranges from 0.6 to 180000.

For most (14/15) cases where Occurrence wins, the number of branches is within 10% of Baseline-Cell-Sum: the EGCC propagation is ineffective. However, even with almost the same size of search tree, EGCC slows down the solver by only 1.66 times or less.

6. Experimental Conclusions

In this section I summarize the most important outcomes of the experiments.

6.1. The basic algorithm for pruning target variables

First of all, there are two published algorithms for enforcing GAC-On-X: Régin's algorithm and Quimper's algorithm. Quimper's algorithm has a tighter worst-case bound, and therefore seems to be a more attractive choice. However, I found Régin's algorithm to be more efficient by a substantial margin (1.62 times faster on average).

The second stage of Régin's algorithm (i.e. Tarjan's algorithm) appears to be much more expensive than the first, based on solver profiling. This counterintuitive observation may inform further optimizations, and explains why Quimper's algorithm is less efficient than Régin's.

Despite the experimental findings it is possible that Quimper's algorithm will out-perform Régin's on very large constraints. However given the size of problems that can be solved in real life, and the small asymptotic difference between the two algorithms, the constants are more important than asymptotic behaviour.

Also, it is not clear which maximum flow algorithm should be used with Régin's algorithm. Only one was experimented with: Ford-Fulkerson with breadth-first search. Depth-first search and Dinic's algorithm are other possibilities that cannot be ruled out.

6.2. Optimizations to the basic algorithm

The results show there is huge benefit from using the following optimizations: using a priority queue and running EGCC at a low priority; incremental matching; incremental graph maintenance; and dynamic partitioning. The results on these optimizations are substantial 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.

It is not possible to give a definitive order of importance of these optimizations, because the experiments were cumulative. However, it seems likely that the priority queue is by far the most important, and among the others dynamic partitioning is particularly important because it showed the most benefit for the largest and hardest instances.

Using the transpose graph to compute the maximum flow was not measurably faster, although it was shown to be using fewer CPU instructions by profiling.

Some of the results depend on the context in which EGCC is used. In particular, internal dynamic triggers (IDT) were not of much benefit on the benchmarks (and when combined with dynamic partitioning, actually slow the solver down). In our benchmarks, the target domains are small: in all cases, smaller than or equal the number of target variables. IDT is expected to work best when target domains are large, and hence the proportion of important values is small. Therefore, IDT could be considered if an EGCC constraint will be used with very large domains, to ameliorate the overheads in this case.

6.3. Algorithms for pruning the cardinality variables

The findings for pruning cardinality variables are straightforward. Adding an implied sum constraint over the cardinality variables has a low overhead and is frequently helpful in reducing the number of branches or the time to reach the fixpoint.

Using Quimper's flow-based algorithm is expensive, slowing down a substantial number of the benchmarks, therefore I cannot recommend it as a default choice. However it is more powerful than the simple algorithm with the sum constraint, solving two extra instances within the half-hour time limit.

6.4. Comparing against Occurrence

Baseline-Cell-Sum never takes more than twice as long as Occurrence to solve any of the benchmarks, and is typically orders of magnitude faster. This is encouraging, and suggests that Baseline-Cell-Sum could perhaps be used as a default choice by an automated modelling assistant.

6.5. Other levels of consistency

In this paper I have focused exclusively on GAC-On-X, and this allowed an extensive study of algorithms for that case. However, I have not compared GAC-On-X against bounds or range consistency, so can offer no conclusions on the relative merits of different levels of consistency. This would be a very interesting avenue of further work.

7. Conclusions

I have presented an extensive survey of propagation methods for the EGCC constraint, studying the pruning of both target variables and cardinality variables, surveying many methods from the literature and presenting some methods that have not been previously reported.

I focused on generalized arc-consistency for the target variables (GAC-On-X) and evaluated two basic algorithms from the literature along with five optimizations found in the literature, and two novel optimizations. In each case I have reported on their implementation and given an empirical analysis of their behaviour. While it was impossible to experiment with every possible combination of optimizations, I took care to compare each optimization against an appropriate baseline method, and to avoid straw men. Particular attention was paid to evaluating *combinations* of optimizations, which is (naturally) not usually a feature of papers that propose optimizations. The experiments presented here comprise easily the deepest experimental analysis of GAC-On-X algorithms. Based on them, I was able to conclude that some optimizations are key and others are less generally useful.

I would like to draw particular attention to the results with dynamic partitioning, a novel generalization of an optimization for AllDifferent [6]. With EGCC dynamic partitioning was 1.56 times faster on average, with a maximum of 5.6 times. The largest gains were seen on the most difficult instances where the solver timed out. The gain for EGCC is less pronounced than for AllDifferent [6], albeit on entirely different benchmarks, and with a different combination of other optimizations.

For the best combination of optimizations, I found a mean improvement of more than 4 times in runtime over a careful but unoptimized implementation of Régin's algorithm. This confirms that optimizations are an essential part of a practical implementation of EGCC.

Regarding the cardinality variables, I was able to confirm that the implied sum constraint used by Gecode is indeed valuable, and also that the stronger flow-based pruning algorithm given by Quimper et al [18] can also be valuable, since it solves more instances within a time limit than either other method.

Finally, a fast variant of EGCC is typically orders of magnitude better than a set of occurrence constraints. Even when EGCC propagation was not effective, it slowed the solver down by only 1.66 times or less in the experiments.

Acknowledgements

I owe a debt to many people for helpful discussions, in particular Ian Gent, Ian Miguel, and Guido Tack. I would also like to thank Ian Gent and Ian Miguel for comments on a draft of this paper. I thank Chris Jefferson for helpful discussions and for pointing out that EGCC cannot be entailed, and Andrea Rendl for use of Tailor [7] for generating benchmarks. This work was supported by EPSRC grants EP/E030394/1 and EP/H004092/1.

References

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

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

[3] Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11:315–333, 2006.

[4] 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.

[5] 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.

[6] Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, 2008.

[7] 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.

[8] Emmanuel Hebrard. Mistral, a Constraint Satisfaction Library. In *Proceedings of the Third International CSP Solvers Competition (CPAI08)*, pages 31–40, 2008.

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

[10] 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.

[11] Sophie Huczynska, Paul McKay, Ian Miguel, and Peter Nightingale. Modelling equidistant frequency permutation arrays: An application of constraints to mathematics. In *Proceedings CP 2009*, pages 50–64, 2009.

[12] 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.

[13] Irit Katriel and Sven Thiel. Complete bound consistency for the global cardinality constraint. *Constraints*, 10(3):191–217, 2005.

[14] Francoise Laburthe. Choco: a constraint programming kernel for solving combinatorial optimization problems. http://choco.sourceforge.net/.

[15] 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.

[16] Peter Nightingale. Are adjacency lists worthwhile in alldifferent? Technical Report CIRCA Preprint 2009/20, University of St Andrews, 2009.

[17] Claude-Guy Quimper. *Efficient Propagators for Global Constraints*. PhD thesis, University of Waterloo, 2006.

[18] Claude-Guy Quimper, Alejandro López-Ortiz, Peter van Beek, and Alexander Golynski. Improved algorithms for the global cardinality constraint. In *Proceedings 10th Principles and Practice of Constraint Programming (CP 2004)*, pages 542–556, 2004.

[19] Claude-Guy Quimper, Peter van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Proceedings 9th Principles and Practice of Constraint Programming (CP 2003)*, pages 600–614, 2003.

[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] Claude-Guy Quimper and Toby Walsh. Global grammar constraints. In *Proceedings of CP-2006*, pages 751–755, 2006.

[22] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[23] 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.

[24] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings 13th National Conference on Artificial Intelligence (AAAI 96)*, pages 209–215, 1996.

[25] Jean-Charles Régin and Jean-François Puget. A filtering algorithm for global sequencing constraints. In *Proceedings 3rd Constraint Programming (CP 97)*, pages 32–46, 1997.

[26] Marko Samer and Stefan Szeider. Tractable cases of the extended global cardinality constraint. In *Proc. Fourteenth Computing: The Australasian Theory Symposium (CATS 2008)*, pages 67–74, 2008.

[27] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(1), 2008.

[28] Christian Schulte and Guido Tack. Weakly monotonic propagators. In *Principles and Practice of Constraint Programming (CP 2009)*, pages 723–730, 2009.

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

[30] Helmut Simonis. A hybrid constraint model for the routing and wavelength assignment problem. In *Proceedings of CP-2009*, pages 104–118, 2009.

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

[32] Robert Endre Tarjan. *Data Structures and Network Algorithms.* SIAM, 1983.

[33] Willem-Jan van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. Revisiting the sequence constraint. In *Proceedings 12th Principles and Practice of Constraint Programming (CP 2006)*, pages 620–634, 2006.

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

[35] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. A tool suite for simulation based analysis of memory access behavior. In *Proceedings of the 4th International Conference on Computational Science (ICCS 2004)*, 2004.