Extension and evaluation of the global cardinality constraints functionality of the Gecode open source toolkit

Ioannis Papatsoris
Supervisor: Panagiotis Stamatopoulos

Our contribution

- Expand open source constraint solver Gecode with global cardinality constraints
- Global Cardinality With Costs
- Symmetric Global Cardinality
- Experiment with different implementation choices for the constraints to optimize performance
- Discover under which circumstances they outperform Gecode's default approach

Constraint Programming

- An Artificial Intelligence methodology that aims to solve complex problems
- Model a problem mathematically using variables, values and constraints
- Variables: entities of the problem
- Values: alternative decisions for each variable
- Constraints: relations that restrict feasible value combinations
- Solution: assignment of a value to each variable, satisfying the constraints

Arc consistency

- Not all combinations of variables and values necessarily form a solution
- Constraints often attempt to remove inconsistent values, for which they can infer with certainty that they cannot participate in any solution
- Example:

$$X = \{7, 8\}$$
$$Y = \{6, 10\}$$
$$X < Y$$

If we assign Y = 6 then there is no legal value for X, so 6 is an inconsistent value for Y and can be pruned

Constraint Solvers

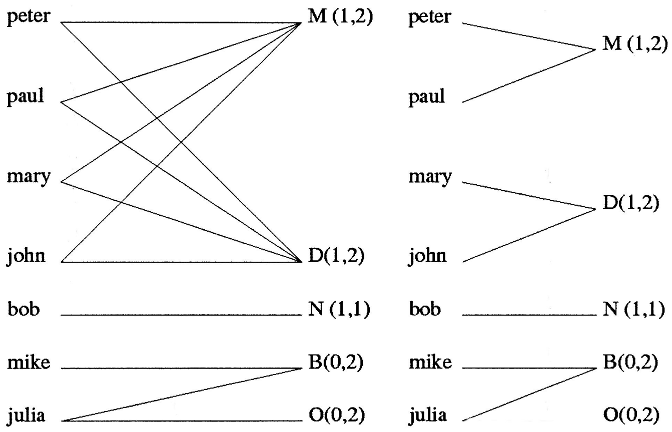
- Platforms that provide the necessary environment to develop and execute constraint programs efficiently
- Declare problem properties, rather than algorithmic steps to solve, solver handles the rest
- Can be used to approach NP-Hard problems like crew scheduling, timetable creation, resource management, car sequencing, protein structure prediction, and more

Constraint Solvers: Search

- Solver enumerates different value combinations
- For each variable, create 2 choices: either assign it with a value, or remove it from its domain
- During each step, the constraints are checked for consistency
- When all variables are assigned, if the constraints are satisfied, we have a solution
- If at any point a constraint is not consistent, search backtracks its choices and takes an alternative decision
- Arc consistency can be used to reduce search space and detect failures early

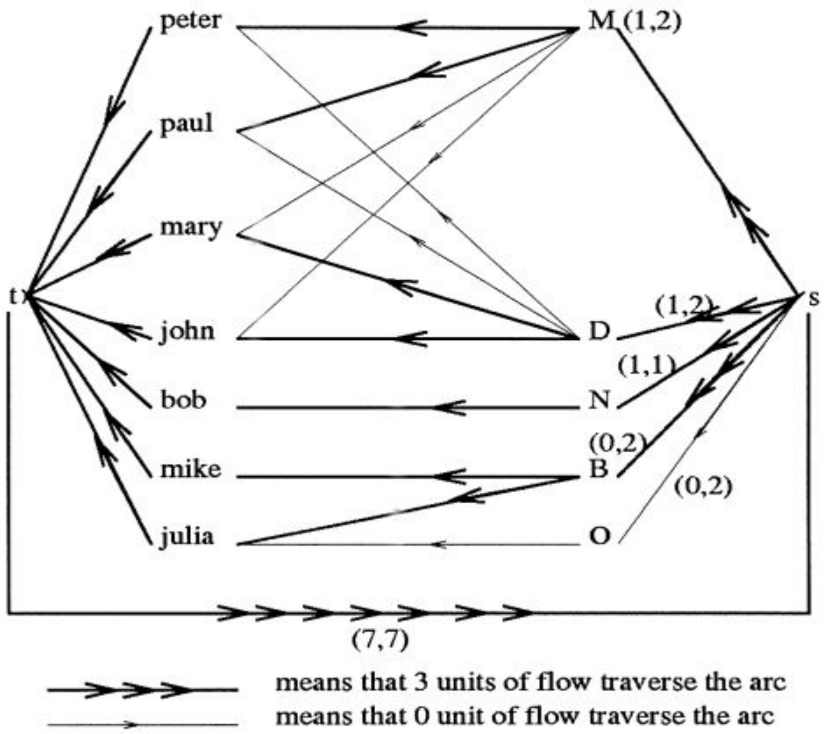
Global Cardinality Constraint (GCC)

- Restricts the value occurrences among a collection of variables, to be between certain bounds
- Each value is associated with a lower and upper occurrence limit
- Applications: car sequencing, sports scheduling
- Already exists in Gecode



Algorithm for GCC

- Create variable-value network
- Use flow theory to map a flow to a solution



Algorithm for GCC

- Start with no flow
- Choose an arc that violates flow restrictions (e.g. S->M)
- Find a <u>path</u> from M to S, send flow along it
- This way we respect the flow conservation law
- Repeat until all flow restrictions are met

Algorithm for GCC: Residual Graph

- Work on Residual Graph instead; a graph that remembers flow history and allows us to take back actions
- Identical to the variable-value network, but for each edge A->B that has flow, include mirror edge B->A
- Traversing mirror edge through a path means that we revert our choice and remove flow from A->B

Arc consistency for GCC

- To assign a value to a variable, we need to send flow through its edge
- To do this, there needs to be a path in the residual graph from the variable to the value
- Variables and values that belong to different SCC, cannot be assigned to each other, because there will never be a path connecting them
- For arc consistency, find SCC, prune variable-value pairs that belong to different one

Global Cardinality Constraint With Costs (COST-GCC)

- Just like GCC, but associates a cost to each variable-value assignment
- The sum of the assigned costs should be less than a given cost bound
- Application: costs can signify preference values in scheduling problems. Sum of preferences should be less than a bound, to ensure quality

COST-GCC vs GCC

- Find <u>shortest paths</u> when sending flow, to achieve a min-cost flow
- Total cost of min-cost flow must be less than bound, if not there is no solution
- Arc consistency can do more: prune variable-value pairs that belong to different SCC like before, **or** that belong in the same one but assigning them would result in a min cost flow which violates the cost upper bound
- In practice, instead of finding SCC, we need to find shortest paths from each variable to each candidate value in the graph

COST-GCC implementation

- Residual graph contains negative costs; to use Dijkstra’s algorithm, original paper suggests transforming them to reduced costs, which are non-negative
- Instead, we keep them negative and use **Bellman-Ford’s** algorithm. Worse complexity but faster in practice. No overhead to maintain reduced costs
- As values get pruned from search, edges are deleted. But we need to be able to backtrack to previous states of the graph efficiently
- Naive way: copy the entire graph on each step, so we can revert to previous states and undo edge deletions
- Better way: use a **smart structure** to represent the neighbors of each node, that minimizes copying overhead and allows fast recovery of previously deleted edges

COST-GCC implementation

Structure holding edges of a particular node:
Array size: 4
Array contents: 7 8 2 1
Pos hash table: 7:0 8:1 2:2   1:3

Delete neighbor 8:

Array size: **3**	**Decrement size**
Array contents: 7 **1** 2 ~~8~~	**Swap element to be deleted with last one**
Pos hash table: 7:0 8:**3** 2:2 1:**1**	**Update positions hash table**

Backtrack deletion:

Array size: **4**	**Backtrack previous size number**
Array contents: 7 1 2 8	

COST-GCC branching

- The nature of the algorithm gives us access to valuable branching heuristic information
- Each time COST-GCC is checked for consistency, it calculates a feasible flow internally, which maps to a solution. We can guide the search and prioritize branching first on values which are known to form a solution, based on the calculated flow.
- Dramatically reduces failures during search

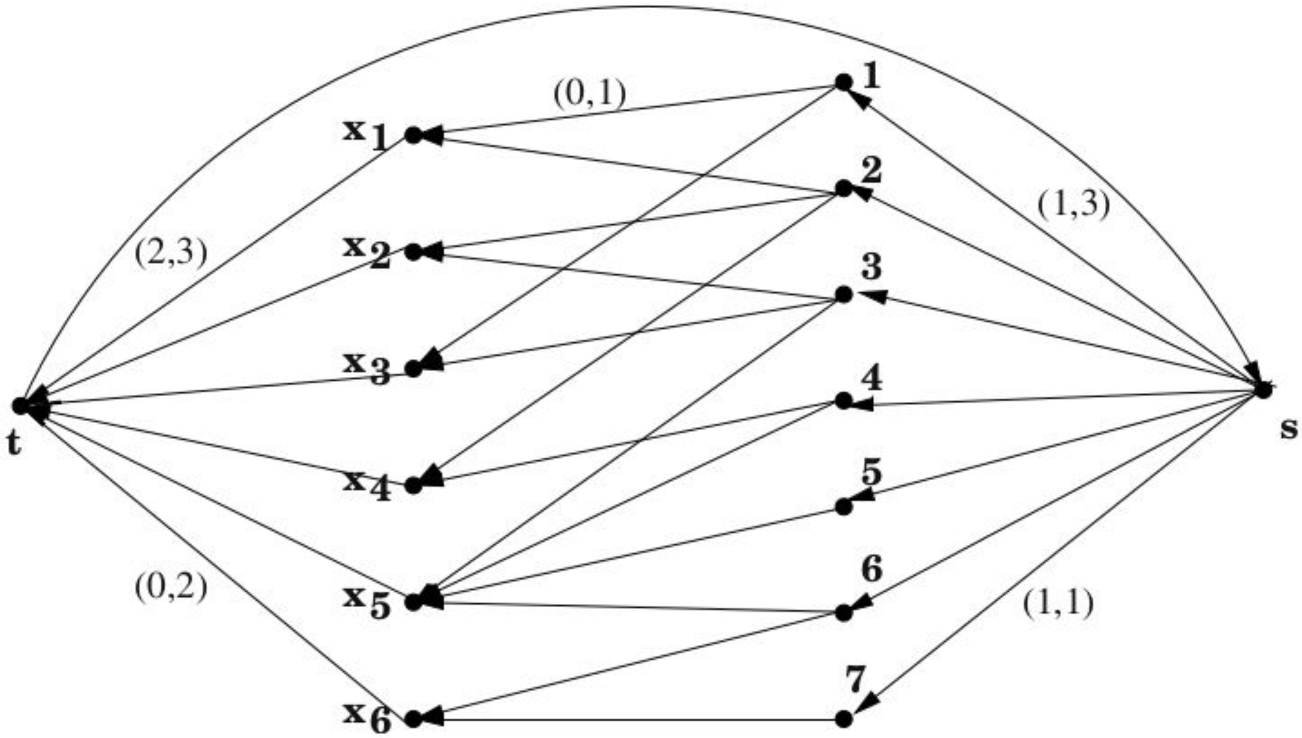
Symmetric Global Cardinality Constraint (SYM-GCC)

- Just like GCC, but on set variables
- Each variable has lower and upper bound restrictions on its set cardinality, on top of the GCC limits on the frequency of values
- Applications in scheduling: Workers and tasks, each worker needs to complete a minimum and maximum number of tasks (set cardinality restriction), and each task requires a minimum and maximum number of workers (value restriction)

SYM-GCC vs COST-GCC

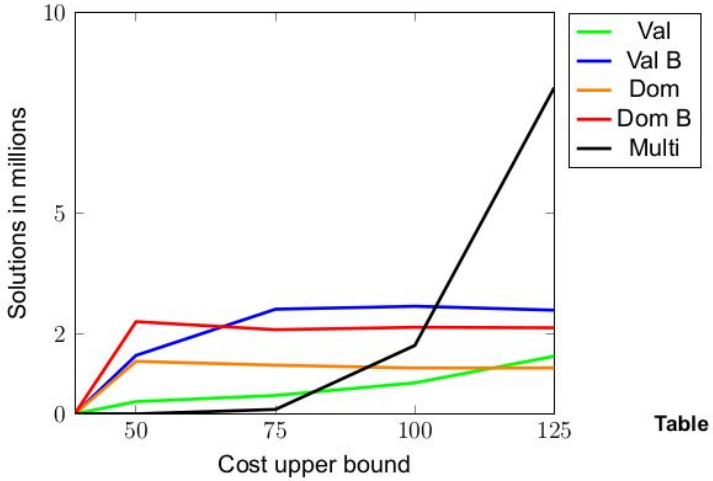
- Easier than COST-GCC, similar to GCC
- No costs; we care about normal paths and not shortest ones (DFS instead of Bellman-Ford)
- Arc consistency: Tarjan's algorithm to find SCC instead of finding specific shortest paths
- Use same smart backtracking structure as in COST-GCC

SYM-GCC Variable-Value graph



COST-GCC Evaluation

Figure 7: Number of solutions for increasingly higher cost upper bound for instance *low-var-big-val-sparse*

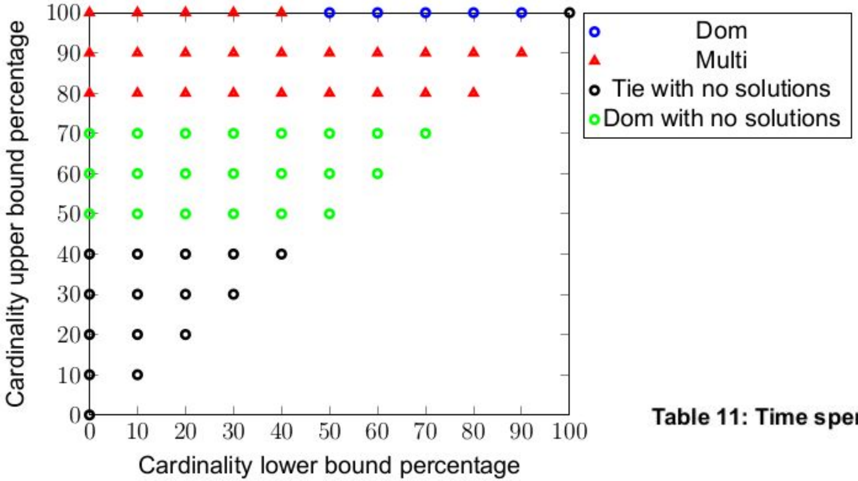


	Val	$ValB$	Dom	$DomB$	$Multi$
br17	1	$3m26s$	$10s$	$5s$	$10ms$
ftv33	254	61	$2.1s$	$0.6s$	269
bays29	892	392	$1m35s$	$15s$	87
berlin52	11879	2053	3030	989	2287

Table 9: Distance to optimal solution with 5min cutoff for different city instances, or time spent in case the optimal solution was found

SYM-GCC Evaluation

Figure 9: Comparison of symgcc versus Multi for instance *big-var-low-val-sparse*, for different cardinality bound percentages



	Val	Dom	$Multi$
6	$71ms$	$3ms$	$12ms$
8	$2m53s$	$3.9s$	–
10	–	$6m22s$	–

Table 11: Time spent to find a solution to the Sports Tournament Scheduling Problem for different number of teams

Conclusions

- We implemented 2 constraints that did not exist in Gecode, experimented with alternative implementation choices to optimize performance, and discovered under which circumstances they outperform Gecode’s default way to simulate them
- More implementation ideas and experiments are discussed in thesis
- Future work:
    - Pull request on Github to contribute to Gecode’s API
    - Implement more constraints, for instance Symmetric Global Cardinality With Costs