Nested Constraint Programs

Geoffrey Chu and Peter J. Stuckey

National ICT Australia, Victoria Laboratory,
Department of Computing and Information Systems,
University of Melbourne, Australia
{gchu,pjs}@cis.unimelb.edu.au

Abstract. Many real world discrete optimization problems are expressible as nested problems where we solve one optimization or satisfaction problem as a subproblem of a larger meta problem. Nested problems include many important problem classes such as: stochastic constraint satisfaction/optimization, quantified constraint satisfaction/optimization and minimax problems. In this paper we define a new class of problems called nested constraint programs (NCP) which include the previously mentioned problem classes as special cases, and describe a search-based CP solver for solving NCP's. We briefly discuss how nogood learning can be used to significantly speedup such an NCP solver. We show that the new solver can be significantly faster than existing solvers for the special cases of stochastic/quantified CSP/COP's, and that it can solve new types of problems which cannot be solved with existing solvers.

1 Introduction

An *aggregator constraint* takes the form: $y = agg([f(x_1, \ldots, x_n, z_1, \ldots, z_m) \mid z_1, \ldots, z_m \text{ where } C(x_1, \ldots, x_n, z_1, \ldots, z_m)])$ where agg is an aggregator function such as **sum**, **max**, **min**, **and**, **or**, f is a function which we will call the local function, c is a (set of) local constraint(s), x_i are some input variables, y is an output variable, and z_i are some local variables.

Aggregator constraints are an extremely flexible and powerful modelling construct, especially when we allow them to be nested inside each other. Problems such as constraint satisfaction/optimization problems, stochastic constraint satisfaction/optimization problems, quantified constraint satisfaction/optimization problems, bi-level and multi-level programming, and many others, can be expressed using aggregator constraints. Unfortunately, most of these problem classes, and the solvers designed for them, only support a very restricted subset of aggregator constraints. For example CP solvers typically cannot handle aggregator constraints natively at all, and rely on some sort of *unrolling* procedure to convert them into primitive constraints first.

An aggregator constraint can be unrolled by eliminating the local variables and local constraints in the aggregator. If we can statically find the set of local solutions S to the constraint $C(x_1, \ldots, x_n, z_1, \ldots, z_m)$ (either independent of x_i or if the x_i are fixed), then for each $\theta \in S$, we create a variable $a[\theta]$ and post the constraints $a[\theta] = f(\theta)$, then post the constraint $y = agg([a[\theta] \mid \theta \in S])$. This

completely eliminates the need to handle those local variables and constraints during solving, but at a potential cost of creating exponentially many variables and constraints.

Example 1. Suppose we have variable arrays p and q. Suppose we have aggregator constraint $y = \mathbf{max}([p[z] + z \times q[z] \mid z \textit{ where } z \in \{1, 2, 3\})$. We can unroll this to: $y = max([a[1], a[2], a[3]])$, $a[1] = p[1] + q[1]$, $a[2] = p[2] + 2 \times q[2]$, $a[3] = p[3] + 3 \times q[3]$. □

Such an approach can handle many common CSP problems, and can also be used to convert stochastic CSP's and quantified CSP's into normal CSP's which can be solved using standard CP solvers. For example, scenario-based methods for solving stochastic CSP's [1] eliminate the stochastic variables in order to convert the problem into a CSP.

However, there are significant problems with this approach. Firstly, in general, it may not be possible or efficient to calculate the set of local solutions S statically. E.g., if the input variables x_i are not fixed at compile time, then it may not be possible at all, or if we have complex constraints in c (like other aggregator constraints), it may take exponential time just to check the satisfiability of an assignment. Secondly, if there are many nested aggregator constraints, then such unrolling could create an exponential number of variables and constraints, causing the solver to run out of memory. For example, in a k-stage stochastic CSP, we have k nested **max** and weighted **sum** aggregators. If each stage had $O(S)$ scenarios and we were to unroll all the weighted sum aggregator constraints by eliminating their local variables, then we end up with $O(S^k)$ variables and constraints which could easily cause the solver to run out of memory. Thirdly, not all the terms in the aggregation are necessarily relevant, especially with aggregators like **max**, **min**, **and**, **or**, etc, where evaluating one term may mean that other terms do not need to be evaluated or do not need to be evaluated fully.

An alternative method for handling aggregator constraints is to keep the local variables and constraints, and dynamically calculate the local solutions to the aggregator constraint during solving, rather than statically at compile time. Such search-based approaches have been used in stochastic CSP/COP solvers [2], in quantified CSP/COP solvers [3], and in quantified Boolean formulae solvers, e.g., [4, 5]. In this paper we go further than these works by defining a much more general class of problems which we will call *nested constraint programs* (NCP's). Rather than only allowing linear aggregation structures where the output of one aggregator is immediately used as the function for the next as in stochastic CSP/COP's and quantified CSP/COP's, we represent the output of aggregators with a variable and allow these variables to be used in an arbitrary manner in the parent context. This means that they can be used in constraints, or as part of some complex expression for the parent's local function. It also means that we can have multiple aggregator constraints in the local constraints of another aggregator constraint, and thus we can have tree-like quantification structures. NCP's include stochastic CSP/COP, quantified CSP/COP/COP+ and many more as special cases, but can model problems which do not fit in any of these subclasses. We describe a new CP-based solver for solving NCP's, and briefly describe how to apply nogood learning in such a solver.

The contributions of this paper are:

- An expressive framework for nested constraint programs (Section 3).
- A propagation based solver architecture that supports this class of problems (Section 4).
- Experiments showing that the resulting system is highly competitive with existing solvers for specialized subclasses of NCP (Section 5).

2 Preliminaries

A *valuation*, θ, is a mapping of variables to values, denoted $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$. Let $vars(\theta) = \{x_1, \ldots, x_n\}$. We can apply a valuation to a variable $\theta(x_i)$ to return the value d_i, and extend application of valuations θ to arbitrary expressions involving $vars(\theta)$ in the obvious way.

A *constraint*, c, is a set of valuations over a set of variables $vars(c)$. A valuation θ is a *solution* of c if $\{x \mapsto \theta(x) \mid x \in vars(c)\} \in c$. A valuation θ is a solution of a set of constraints C if it is a solution for each $c \in C$. We write $c_1 \models c_2$ if every solution of c_1 is a solution of c_2.

A *literal* is a unary constraint (we can restrict to the forms $x = d, x \neq d, x \geq d, x \leq d$), or *false*. A *domain* D is a conjunction of literals over $vars(D)$. We use notation $D(x) = \{\theta(x) \mid \theta \text{ is a solution of } D\}$. We use *range notation* $[\, l \,..\, u \,] = \{d \mid l \leq d \leq u\}$. A *singleton domain* is one where $|D(x)| = 1, x \in vars(D)$, and we let $\theta_D = \{x \mapsto d_x \mid x \in vars(D), D(x) = \{d_x\}\}$ in this case.

A *propagator* $p(c)$ for constraint c is an inference algorithm, it maps a domain D to a conjunction of literals $p(c)(D)$, where $D \wedge c \models p(c)(D)$. We shall sometimes treat this conjunction as a set. We assume each propagator is *checking*, that is if $\forall x \in vars(c).|D(x)| = 1$ then $p(c)(D) = \emptyset$ if θ_D is a solution of c and $\{false\}$ otherwise.

In *lazy clause generation (LCG)* solvers [6, 7] propagators are also required to return explanations for each new consequence $l \in p(c)(D)$, that is an explanation clause $e \equiv l_1 \wedge \cdots l_n \rightarrow l$ where $\forall 1 \leq i \leq n, D \models l_i$ and $c \models e$. LCG solvers, like SAT solvers, create an implication graph, where every new consequence is attached to a reason. On failure this used to create a *nogood* by repeatedly replacing literals in the explanation of failure until only one literal that became true after the last decision remains. This nogood is guaranteed to generate new propagation information. See [5] for more details.

3 Aggregators and Nested Constraint Programs

An *aggregator function* is a function which maps a multiset (list) of values to a single value by performing some sort of aggregation over them, e.g., by summing over them, or taking the maximum, etc. Aggregators are functions on multisets, they cannot make use of the order of elements in the list they operate on. They may be partial functions.

An *aggregator constraint* is of the form: $y = agg([f(x_1, \ldots, x_n, z_1, \ldots, z_m) \mid z_1, \ldots, z_m \text{ where } C(x_1, \ldots, x_n, z_1, \ldots, z_m)])$ where agg is an aggregator function,

f is the *local function* of the aggregator constraint, and C is a set (or conjunction) of *local constraints* of this aggregator constraint. We assume the local function f is total in its inputs, if not we can add constraints to C to ensure it is total for all possible local solutions, thus implementing the relational semantics [8]. The scope of this aggregator constraint is $vars(a) = \{y, x_1, \ldots, x_n\}$. Given aggregator constraint a, let $ovar(a) = y$ be the *output variable*, $ivars(a) = \{x_1, \ldots, x_n\}$ be the *input variables*, $lvars(a) = \{z_1, \ldots, z_m\}$ be the *local variables*, and $lcons(a) = C$.

The solutions of an aggregator constraint $c \equiv y = agg([f(x_1, \ldots, x_n, z_1, \ldots, z_m) \mid z_1, \ldots, z_m \textit{ where } C(x_1, \ldots, x_n, z_1, \ldots, z_m)])$ are defined inductively on the depth of nesting. Let Θ be the solutions of the constraint C, then the solutions of c are

$$\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n, y \mapsto agg([f(d_1, \ldots, d_n, \theta(z_1), \ldots, \theta(z_m) \mid \theta \in \Theta, \theta(x_i) = d_i])\}$$

for all $(d_1, \ldots, d_n)$ where $\exists \theta \in \Theta . \forall 1 \leq i \leq n . \theta(x_i) = d_i$. If C includes no aggregator constraints then this definition is self-contained, otherwise we can determine the solutions of C by induction (since the depth of nesting is 1 less) and use them to define the solutions of c. Note that an aggregator constraint may have no solutions if the aggregation function is not defined on the resulting multiset of solutions, e.g. $y = \textbf{average}([])$.

Example 2. Suppose we have $y = \textbf{sum}([z_1 \mid z_1, z_2 \textit{ where } z_1, z_2 \in [1..3], z_1 + z_2 \leq x])$. Then given a particular value of x, we find all solutions to z_1, z_2 which satisfy $z_1, z_2 \in [1..3], z_1 + z_2 \leq x$, and sum over the z_1 values of those solutions in order to calculate y. So for example, this constraint would allow the tuples: $(x, y) \in \{\ldots, (0,0), (1,0), (2,1), (3,4), (4,10), (5,15), (6,18), (7,18), \ldots\}$. □

Some commonly used aggregators are **sum**, **product**, **min**, **max**, **and**, **or**, **average**, **stddev**, **variance**, whose definitions are already well known. Note that we have not restricted the types of the variables or the input or output arguments to the aggregator functions. This means for example, we can define aggregator functions which take in a list of tuples as arguments, or which return a tuple as the output, etc. For example, weighted average can be defined on a list of pairs where the first element is the weight and the second element is the value and it returns a single value as output. Similarly, we can extend **min** and **max** to take a list of tuples as argument and use the lexico-graphical ordering to return the smallest or largest tuple as the return value.

A *nested constraint program* (NCP) consists of a single aggregator constraint with no input variables: e.g. $y = agg([f \mid z_1, \ldots, z_m \textit{ where } C(z_1, \ldots, z_m)])$ The goal of an NCP is to determine the value of the output variable of the top-level aggregator constraint. The power of NCPs arise from the fact that the local constraints of one aggregator constraint can contain other aggregator constraints. Thus in general, we can have a nested structure where we have a tree of aggregator constraints, each with its own local variables and constraints, and where each aggregator constraints is a local constraint of its parent aggregator constraint.

Example 3. In the simple production planning problem studied in [2], in each stage, we can choose to produce 0 or more books. After production, there is a stochastic demand for books between 100 and 105 with equal probabilities for

each. There are soft constraints enforcing that the available stock be sufficient to satisfy the demand. The problem is to find a policy whose expected satisfiability is above a certain threshold α. We can model a 3 stage instance as follows:

$$
\begin{array}{l}
r = \mathbf{or}([m_1 \geq \alpha \mid m_1 \ \mathit{where} \\
\quad m_1 = \mathbf{max}([a_1 \mid s_1, p_1, a_1 \ \mathit{where} \ s_1 = 0 \wedge \\
\quad\quad a_1 = \mathbf{average}([\mathtt{bool2int}(s_1 + p_1 \geq d_1) \times m_2 \mid d_1 \in [100..105], m_2 \ \mathit{where} \\
\quad\quad\quad m_2 = \mathbf{max}([a_2 \mid s_2, p_2, a_2 \ \mathit{where} \ s_2 = s_1 + p_1 - d_1 \wedge \\
\quad\quad\quad\quad a_2 = \mathbf{average}([\mathtt{bool2int}(s_2 + p_2 \geq d_2) \times m_3 \mid d_2 \in [100..105], m_3 \ \mathit{where} \\
\quad\quad\quad\quad\quad m_3 = \mathbf{max}([a_3 \mid s_3, p_3, a_3 \ \mathit{where} \ s_3 = s_2 + p_2 - d_2 \wedge \\
\quad\quad\quad\quad\quad\quad a_3 = \mathbf{average}([\mathtt{bool2int}(s_3 + p_3 \geq d_3) \mid d_3 \in [100..105]])])])])])])])
\end{array}
$$

Example 4. Consider the 2-player Nim-Fibonacci game [10]. The game starts with n matches. The first player may take between 1 to $n-1$ of the matches. Thereafter, the turns alternate and the current player may take between 1 to $2k$ of the matches where k is the number of matches taken by the previous player. The player who takes the final match wins. The problem is to find out for each n whether the first player has a winning strategy. It has the interesting property that the first player has a winning strategy iff n is not a Fibonacci number. The problem can be modelled as follows. Given turn i, let r_i be the number of matches left, l_i be the maximum number of matches that can be taken during that turn, t_i the actual number of matches taken, and w_i whether there is a winning strategy from that position.

$$
\begin{array}{l}
w_1 = \mathbf{or}([l_1 \geq r_1 \vee \neg w_2 \mid w_2, l_1, r_1, t_1 \ \mathit{where} \\
\quad l_1 = n - 1 \wedge r_2 = n \wedge 1 \leq t_1 \leq l_1 \wedge \\
\quad w_2 = \mathbf{or}([l_2 \geq r_2 \vee \neg w_3 \mid w_3, l_2, r_2, t_2 \ \mathit{where} \\
\quad\quad l_2 = 2 \times t_1 \wedge r_2 = r_1 - t_1 \wedge 1 \leq t_2 \leq l_2 \wedge \\
\quad\quad w_3 = \mathbf{or}([l_3 \geq r_3 \vee \neg w_4 \mid w_4, l_3, r_3, t_3 \ \mathit{where} \\
\quad\quad\quad l_3 = 2 \times t_2 \wedge r_3 = r_2 - t_2 \wedge 1 \leq t_3 \leq l_3 \wedge \\
\quad\quad\quad \ldots \\
\quad\quad\quad w_n = \mathit{true}]) \ldots])
\end{array}
$$

The ability to model problems using tree-like quantification structures rather than the linear quantification structure used in stochastic CSP and quantified CSP allows certain kinds of optimisations.

Example 5. Consider a stochastic scheduling problem with precedence and non-overlap constraints C on n tasks where we fix the (array of) start times $\bar{s}$ within the makespan $[\,0\,..\,m\,]$, written as $\bar{s} \in \overline{[\,0\,..\,m\,]}$, but then each task duration d_i can then independently be one of three values $L_i = \{f_i, r_i, w_i\}$ fast, regular or slow, and we need to pay recourse $\mathit{recourse}_c(s_{c_1}, s_{c_2}, d_{c_1}, d_{c_2})$ for the violation of each constraint $c \in C$ involving at most two tasks numbered c_1 and c_2. Given n tasks there are 3^n scenarios. The natural stochastic model is

$$
\begin{array}{c}
u = \mathbf{min}([\mathbf{average}([\mathbf{sum}([\mathit{recourse}_c(s_{c_1}, s_{c_2}, d_{c_1}, d_{c_2}) \mid c \in C]) \\
\mid \bar{d} \in \bar{L}]) \mid \bar{s} \in \overline{[\,0\,..\,m\,]}])
\end{array}
$$

where we find a schedule (start times) $\bar{s}$ which minimizes the expected recourse cost, over all possible durations $\bar{d} \in \bar{L}$ scenarios The problem is there are 3^n

scenarios and hence evaluating a schedule is $O(|C|3^n)$. But, since the recourse for each constraint $c \in C$ is dependent on at most two stochastic durations d_{c_1} and d_{c_2}, we can model this instead as

$$u = \mathbf{min}([\mathbf{sum}([\mathbf{average}([recourse_c(s_{c_1}, s_{c_2}, d_{c_1}, d_{c_2}) \mid d_{c_1} \in L_{c_1}, d_{c_2} \in L_{c_2}]) \mid c \in C])) \mid \bar{s} \in \overline{[0\,..\,m\,]}])$$

There are at most 9 scenarios for each constraints recourse calculation, hence to evaluate a schedule is $O(|C|)$. Note that such a quantification structure is not supported by stochastic CSP solvers but can be solved with our more generic NCP solver. □

Aggregator functions involving tuples allows us to model things like bi-level programs.

Example 6. The Network Links Pricing problem [3] can be described as follows. The problem is to set the tariffs on the network links in order to maximise the profit of the owner of the links. The $i \in I$ customer (or data movement) will route their d_i data from src_i to snk_i using the smallest possible cost path. Each path has to cross a tolled arc $j \in J$ with cost per unit data t_j. We assume the cost per unit data from src_i to snk_i via j is $c_{i,j}$ for the rest of the network. Hence the cost to the customer of a path through arc j is $(c_{i,j} + t_j) \times d_i$. The income to the network is $t_j \times d_i$. The customer can choose another independent network provider with cost u_i instead of using this network. The problem is determine the toll t_j from some set of possibilities T_j ($\bar{t} \in \bar{T}$) for each arc j to maximise revenue. We assume customers will pick the cheapest link for them, but if there are ties, they will pick the one most profitable for the operator, as the network operator can simply adjust their tariffs by some small ϵ to make that choice the cheapest for that customer. The interesting thing here is that in the subproblem, we need to minimize one quantity, i.e., the cost the customer, but return another value as the result, i.e., the profit to the operator. Such problems cannot be modelled as normal QCOPs, but can be modelled as QCOP+s or as NCPs by using tuples.

$$y = \mathbf{max}([\mathbf{sum}(p * d_i \mid c, p \textit{ where } (c, p) = \mathbf{max}([(-(c_{i,j} + t_j), t_j) \mid j \in J \textit{ where } (c_{i,j} + t_j) \times d_i \leq u_i])) \mid i \in I]) \mid \bar{t} \in \bar{T}])$$

Note that the inner (lexico-graphical) **max** on the pair picks the link with the lowest cost, and among those the one with highest profit, and returns that pair as the return value. □

The greater expressivity of NCP allow us to model all kinds of meta problems where the results or properties of subproblems can be used in constraints or the objective function of the parent problem, or the output of one subproblem can be used as the input to another, etc.

Example 7. Consider a Sudoku problem, given a set of possible clues $\{x_{i_l j_l} = d_l \mid 1 \leq l \leq n\}$, find the smallest subset of these clues such that the resulting Sudoku problem has a unique solution. We can model this as a NCP, using a Boolean variable b_l to indicate whether a clue is used:

$$\begin{aligned} c = \mathbf{min}([\mathbf{sum}([b_l \mid l \in [\,1\,..\,n\,]]) \mid \bar{b} \in \overline{[\,0\,..\,1\,]} \;\; \textit{where} \\ 1 = \mathbf{sum}([1 \mid \bar{x} \in \overline{[\,1\,..\,9\,]} \;\textit{where} \\ \mathbf{and}([b_l \rightarrow x_{i_l j_l} = d_l \mid l \in [\,1\,..\,n\,]]) \wedge \textit{sudoku}(\bar{x})])]) \end{aligned}$$

where $\textit{sudoku}(\bar{x})$ are constraints enforcing that $\bar{x}$ is a solution to a Sudoku problem. □

CSP/COP, stochastic CSP/COP [2], QCSP [11], QCOP, QCOP+ [3], stochastic SAT, MAXSAT, QBF [12], influence diagrams, finite horizon markov decision processes, MPE and MAP queries over stochastic graphical models such as Bayesian nets, and probably many more, are all expressible as NCP. In addition to this, there are many problems expressible as NCP which cannot easily be expressed as any of the previously mentioned problem classes. Thus NCP is a very expressive and generic problem class. In this paper, we are interested in solving NCPs using a modified CP solver. Thus we restrict our attention to *finite NCPs*, i.e., where every local variable is constrained to have a fixed finite initial domain given by explicit local constraints, e.g. $z \in [\,1\,..\,100\,]$.

4 Solving NCP's

In this section, we describe how to solve NCPs. The main idea is to augment a standard CP solver with a new class of propagators which can encapsulate the subproblem modelled by an aggregator constraint. Such a propagator constrains the input and output variables of the aggregator constraint and performs propagation on these variables. The major difference with normal CP propagators is that unlike a normal CP propagator which run a self-contained algorithm to perform propagation, these new propagators may take control of the search engine itself to do some search in order to perform their propagation. However, at the end of their propagation algorithm, they must return the search engine to its former state so that it is as if nothing has happened. Thus these new propagators simply change the domains of variables, just like all other standard CP propagtors, and there is no need to treat them specially in any way. The fact that the set of assignments allowed by the aggregator constraint is defined via aggregating the solutions to a CP problem rather than extensionally or intensionally as in normal CP constraints is irrelevant. As far as the parent subproblem is concerned, the aggregator constraint is just a normal constraint which is satisfied by a particular set of assignments, and we have a propagator which is capable of enforcing the constraint, and thus the parent subproblem can be treated as a completely normal (non-nested) CP problem. This allows us to solve NCPs using CP solvers by simply adding a new kind of propagator.

Recall that there are two main ways to deal with an aggregator constraint. We can either unroll it by eliminating the local variables and local constraints as described in Section 1, or we can post a propagator which can handle it

natively as described above. The first suffers from a potential combinatorial explosion in the number of variable/constraints required, while the second tends to have weaker propagation strength. The key here is to try to find the best tradeoff between propagation strength and time/memory usage. For simple aggregator constraints, we would often get a better tradeoff simply by unrolling them. Whereas for more complicated ones with non-trivial local constraints, the search based approach may give a better tradeoff. In general, we use the following policy: given an aggregator constraint a, if the local solutions of a can be computed at compile time and there are no more than L of them where L is some user defined parameter, and there are no nested aggregator constraints within a, then we unroll a. Otherwise, we leave it as is and post a propagator that can handle it natively. Such a policy ensures that we avoid any sort of exponential blowup in problem size that can occur when we unroll aggregator constraints.

After unrolling, we create a domain object for each variable and a propagator object for each constraint and aggregator. Unlike a normal CP solver where all variables exist in the one existential context, variables in a NCP may belong to different contexts. Each variable is either a local variable to one aggregator constraint or is the root variable. It can also be an input variable of zero or more descendant aggregator constraints. Each local constraint can contain local variables from the same aggregator constraint and also input variables which are local to the ancestor aggregator constraints.

First, we consider a non-aggregator constraint c. For each such constraints c, we create a modified propagator $p(c)$ which is identical to the standard CP propagator, except that it is only allowed to prune values from the domains of the *local variables* of the parent aggregator constraint $y = agg([...])$ to which it belongs. It is *incorrect* in general to prune values from the input variables. The reason for this is that when a local constraint has no solutions, this does not mean that the parent aggregator constraint has no solutions, rather it means that the value of y is given by $agg([])$ since the local problem has no solutions. This does not constrain the input variables, so pruning their domains is incorrect!

Example 8. Consider the following problem: $y = \mathbf{min}([x_2 \mid x_1, x_2 \textit{ where } x_1 \in [\,0\,..\,3\,] \wedge x_2 \in [\,-4\,..\,4\,] \wedge x_2 = \mathbf{sum}([x_3 \mid x_3 \textit{ where } x_3 \in [\,1\,..\,5\,] \wedge x_3 \leq x_1])])$. The local constraint $x_2 = \mathbf{sum}([x_3 \mid x_3 \textit{ where } x_3 \in [\,1\,..\,5\,] \wedge x_3 \leq x_1])$ has the solutions $(x_1, x_2) \in \{\ldots, (-1, 0),\ (0, 0),\ (1, 1),\ (2, 3),\ (3, 6),\ (4, 10),\ (5, 15),\ (6, 15),\ \ldots\}$. This means that the local solutions of the **min** aggregator are $(x_1, x_2) \in \{(0, 0), (1, 1), (2, 3)\}$, giving $y = 0$. Suppose however, we allowed the constraint $x_3 \leq x_1$ to propagate on its input variable x_1. Then at the root, since we have $x_3 \in [\,1\,..\,5\,]$ we can immediately propagate $x_1 \geq 1$, and we have completely pruned off a totally valid local solution ($(x_1, x_2) = (0, 0)$) of the **min** aggregator, leading to an answer of $y = 1$ which is simply wrong. □

For each aggregator constraint a, we create a propagator $p(a)$. This can use the semantics of its aggregator function to propagate domain changes to the output variable based on the domains of its input and local variables. For example, consider an aggregator constraint $a \equiv y = \mathbf{min}([f(z, x) \mid z \textit{ where } c(z, x)])$. Suppose that a has not yet taken control of the search (i.e., x is not yet fixed), but propagation of the local constraints c has already forced a lower bound l on $f(z, x)$. Then we can immediately propagate $y \geq l$, because no matter what x

ends up being set to, any local solution must have local objective value greater than or equal to l.

Secondly, we maintain a copy of each aggregator constraint in A which will propagate in a more complex way. When all the input variables of a are fixed, then a can take control of the search engine and perform search on its local variables in order to calculate the value of the output variable. Different aggregator constraints can use different search strategies. The search strategy can either be defined in the model, or some sort of autonomous or default search can be used. They will have different conditions for when they can yield control of the search engine. For example, an **or** (resp. **and**) aggregator can yield control as soon as a *true* (resp. *false*) solution is found. A **min** aggregator will perform local branch and bound in order to find its output value. The first branching decision that it will make will be of the form $o < k$ where o is the objective variable, and k is either the value of the best solution found so far, or $\max D(y) + 1$ if it has just taken control.[1] If it finds a solution with objective value k, it can propagate $y \leq k$. If the branch and bound decision $o < k$ produces failure, then it can immediately propagate $y \geq k$. It can yield control if it either proves $y \leq \min D(y) - 1$, $y \geq \max D(y) + 1$ or it finds the optimal solution and proves optimality. Similarly, a **sum** aggregator would perform a search to find all of its local solutions to calculate the sum of the local function values. If it proves that $y \leq \min D(y) - 1$ or $y \geq \max D(y) + 1$, it can terminate early.

Pseudo-code for the algorithm is given in Figure 1. We set up an initial domain D for all variables, and set P to be the propagators $p(c)$ for each $c \in C \cup A$. Initially, the root aggregator *root* is in control of the search engine with the call $\mathsf{agg}(root, D, P, A)$. The aggregator constraint sets the variables V as its input and local variables, and invokes search.

Then propagation is performed to fixed point or failure by propagate. Propagation repeatedly chooses a propagator p from the queue Q, calculates a new domain D' then adds all the propagators in P that may need to be recomputed due to changes in the domain computed by $\mathsf{new}(P, D, D', a)$, repeating until the queue is empty. There is a subtle difference with regular CP propagation. If a variable x gets an empty domain, this does not necessarily mean that the last decision made was infeasible. If $x \notin V$ is not a local variable then it simply means that the aggregator a' that introduces x (which must be a descendent of a in the aggregator tree) has no local solution given the decisions of its ancestor aggregators. This means that a' needs to be woken up so that it can propagate $y = agg([])$ where y is its output variable and agg is its aggregator function. Note that a' has to be a descendant of a, because decisions made by a can only cause domain changes to variables belonging to descendants of a.

If we reach propagation fixed point, then we need to check whether any other aggregator constraints become eligible for taking over control of the search. An aggregator a is *eligible* if:

- its not currently suspended and either all its input variables are fixed or one of its local variables has an empty domain, and
- its output variable has not already been fixed by a when it executed earlier on the same fixed inputs or empty domain, and

[1] Assuming y is integer for simplicity of explanation

– its output variable appears in at least one constraint which is not already satisfied.

In the last case, no other constraint cares about the value so we do not need to calculate it. If an aggregator is eligible, it will immediately take over control of the search engine and the aggregator constraint which was previously in control will be suspended until this one returns. If multiple aggregators become eligible at the same time, then we choose one of the aggregators closest to the root in the aggregator tree.

After propagation quiesces there are three cases. If a *local* variable has no solution this indicates failure, hence search backtracks. If all the local variables are fixed then we have discovered a new solution θ. Then the aggregator constraint will do whatever sort of bookkeeping it needs to do to calculate its aggregate value using process_solution. If it determines it now has enough information to determine the final aggregate value or fail, process_solution returns *true* and the search finishes, otherwise it backtracks and continues the search. If propagation neither fails nor succeeds, the aggregator constraint a in control of the search makes a branching decision $c_1 \vee \cdots \vee c_n$ using its branching heuristic branch(a, D), and searches each resulting subproblem. Search continues until either the entire subtree for this subproblem is explored or process_solution detects early termination.

When search finishes the aggregator calculates the result on its output variable and updates the domain accordingly, then yields control to its parent. The algorithm terminates when the root aggregator constraint yields control, at which point, we have calculated the value of its output variable and solved the NCP.

Example 9. Consider the problem of Example 8. We create an initial domain $D(x_1) = [\,0\,..\,3\,]$, $D(x_2) = [\,-4\,..\,4\,]$, $D(x_3) = [\,1\,..\,5\,]$, $D(y) = [\,-\infty\,..\,\infty\,]$. We create propagators for the constraints $x_3 \leq x_1$ and the two aggregator constraints. Execution begins by calling the search on the root **min** aggregator. Propagation uses $x_3 \leq x_1$ to set $D(x_3) = [\,1\,..\,3\,]$ and queisces. Assume the **min** aggregator makes a branching decision on the x_1 variable $x_1 = 0 \vee x_1 \geq 1$ (since the branch and bound decision $x_2 < +\infty \vee x_2 \geq +\infty$ is not useful). Searching on the left branch sets $D(x_1) = \{0\}$ which causes $D(x_3) = \emptyset$ which wakes the **sum** aggregator (with an empty domain for the local variable x_3) which immediately returns setting $D(x_2) = \{0\}$. The **min** aggregator processes the solution $(x_1, x_2) = (0, 0)$, (we will at this stage propagate that $D(y) = [\,-\infty\,..\,0\,]$). Search then tries the right branch where propagation returns the domain $D(x_1) = [\,1\,..\,3\,]$, $D(x_2) = [\,-4\,..\,4\,]$, $D(x_3) = [\,1\,..\,3\,]$ by propagating the constraint $x_1 \geq 1$. Once again the **min** aggregator makes a branching decision $x_1 = 1 \vee x_1 \geq 2$, and taking the left branch sets $D(x_1) = \{1\}$ and $D(x_3) = \{1\}$ waking the **sum** aggregator since its input variable x_1 is fixed. This aggregator finds a single solution $(x_1, x_3) = (1, 1)$ and then sets $D(x_2) = \{1\}$. The **min** aggregator processes the solution $(x_1, x_2) = (1, 1)$ by just throwing it away. Search continues with the right branch where eventually the **min** aggregator finds the remaining solution $(x_1, x_2) = (2, 3)$, and returns $y = 0$.

Note that if we use a cleverer propagator for the **sum** aggregator then at the first propagation step it will set $D(x_2) = [\,0\,..\,4\,]$ since the sum of any number of

```
agg(a, D, P, A)
    let a = agg([o|lvars(a) where lcons(a)])
    search(D, ivars(a) ∪ lvars(a), P, A, {p(c) | c ∈ lcons(a)}, a)
    let Θ be the set of solutions processed
    D := D ∧ ovar(a) = agg([θ(o)|θ ∈ Θ])
    return D

search(D, V, P, A, Q, a)
    D := propagate(D, V, P, A, Q, a)
    if (∃x ∈ V.D(x) = ∅) return false
    if (∀x ∈ V.|D(x)| = 1)
        let θ = {x ↦ d_x | x ∈ V, D(x) = {d_x}}
        return process_solution(θ, a, D)
    else
        {c_1, ..., c_m} := branch(a, D)
        for i ∈ 1..m
            if (search(D, P ∪ Q, A, {p(c_i)}, a))
                return true
        return false

propagate(D, V, P, A, Q, a)
    P := P ∪ Q
    repeat
        while (∀x ∈ V.D(x) ≠ ∅ ∧ ∃p' ∈ Q)
            Q := Q − {p'}
            D' := D ∧ p'(D)
            Q := Q ∪ new(P, D, D', a)
            D := D'
        if (∃a ∈ A.eligible(a))
            A := A − {a}
            D' := agg(a, D, P, A)
            Q := Q ∪ new(P, D, D', a)
            D := D'
    until Q = ∅
    return D
```

Fig. 1. Pseudo-code for evaluating NCPs.

positive values ($x_3 \in [1..3]$) is at least 0. The **min** propagator can also add the bounding constraint $x_2 < 0$ after it finds the first solution. These two together would cause search to terminate immediately once the first solution was found. □

In this paper we only consider computing the result of the NCP, in practice we will may want to know the "policy" of decisions that lead to this result. It is easy enough to expose the values of the local variables of the root aggregator, thus giving the "first-stage" decisions. To record the entire policy of decisions we would need to store a shorthand form of the entire search tree (including nested search trees) analogous to the approach used in QCOP+ [3].

4.1 Complexity

The time and space complexity of the algorithm depends on many things, such as the time and space complexity of the propagators and aggregators, and the search strategy. However, a very large subclass of finite NCP is PSPACE-complete. In particular, consider the subclass where all non-aggregator constraints have a polynomial space propagator (this is true for all commonly used CP constraints) and all aggregator functions require only a polynomial space to compute their output value when the elements of its list argument are fed in one by one (this is true for **sum**, **product**, **min**, **max**, **and**, **or**, **average**, **stddev**, **variance**, but not for aggregators like **median**). This subclass includes QBFs and quantified CSPs and thus is PSPACE-HARD. For such problems, the algorithm is PSPACE-complete.

4.2 Learning for NCPs

Nogood learning [5, 6] significantly improves both SAT and CP solving performance. Similarly, it dramatically improve the efficiency of an NCP solver. Unfortunately we do not have sufficient space to adequately describe how to add nogood learning to an NCOP solver. Instead we will briefly discuss the uses of nogood learning, and some of the issues that arise in implementing it. There are generally two different kinds of nogoods that we can learn: ones which explain a local failure, and ones which explain the return value of a subproblem.

- Nogoods learned within one execution of an aggregator constraint a become new local constraints for a. Just like other local constraints they are only allowed to prune local variables. When we reexecute a with different input variable values, much of the search in a may be repeated, and these local nogoods can substantially reduce this repeated search, similar to the case for inter-problem nogood learning [15].
- We can cache the return value of an execution of an aggregator constraint a using a nogood, e.g. if we run a with input variables fixed to $x_1 = d_1, \ldots, x_n = d_n$ and find that output $y = d$ we can cache this as $x_1 = d_1 \wedge \cdots x_n = d_n \rightarrow y = d$. This nogood prevents us from having to run the aggregator again on the same input values.
  Nogood learning can be better than this however, since it may determine that only some of the input constraints are required to give the result of aggregator a leading to a much more general nogood. Consider for example the aggregator of Example 2, setting $x = 0$ gives $y = 0$, but nogood learning will learn that $x \leq 1 \rightarrow y = 0$. Similarly, $x = 7$ gives $y = 18$, but nogood learning will learn universally that $y \leq 18$.

The challenge for implementing nogood learning in an NCOP solver is to extend the propagators for aggregator constraints to explain their propagation behaviour. To do so we must be able to determine what parts of the input constraints contributed to the result of the aggregator. This requires combining the usual uses of nogoods, to explain why some part of the search *failed*, with explaining why some part of the search *succeeded with a certain value of the local function*. The success explanation part is entirely novel, and is a generalisation of solution analysis in QBF, (see e.g. [12]) where all constraints are clauses and they use specialized heuristics to pick a satisfying literal for each clause.

5 Experiments

Due to the very large range of problem classes that can be modelled as NCPs, it is difficult to compare against the current state of the art in all those problem classes. Instead, we concentrate on the problem classes where CP-based solvers have had some success, namely in stochastic CSP/COP problems and quantified CSP/COP problems. We implemented a NCP solver in Chuffed, a state-of-the-art CP solver that supports nogood learning. Experiments are run on Intel Xeon 2.40GHz processors, with a 1800s timeout. Times are given in seconds. We use an unrolling limit (L in Section 4) of 100. We use input order search and try the

Table 1. Comparison of the search-based NCP solver with learning (learn) and without learning (no-learn) with QeCode (qecode) on the Nim-Fibonacci Problem.

Size	no-learn		learn		qecode	
	fails	*time*	*fails*	*time*	*fails*	*time*
5	2	0.01	2	0.01	23	0.01
10	20	0.01	13	0.01	1310	0.02
15	174	0.01	45	0.01	11116	0.26
20	1438	0.01	101	0.01	56560	1.61
30	62313	0.36	272	0.01	483346	16.24
40	4773553	30.13	727	0.01	—	TO
50	—	TO	1502	0.0	—	TO
100	—	TO	8461	0.21	—	TO
200	—	TO	45227	2.12	—	TO
500	—	TO	414152	55.29	—	TO

smallest value first. We compare against QeCode 2.0 [3], which is a state of the art QCOP+ solver, on the problems that it supports. QeCode does not support floating point variables or weighted sum aggregators. As a result, it is unable to solve stochastic COP's, so we only compare against it on integer and Boolean problems. We also compare against the published results of other systems that are not publicly available.

The Nim-Fibonacci problem is described in Example 4. It is a QCSP and can potentially be unrolled into a CSP by eliminating the universal variables and solved using a CP solver. However, this is not really practical as this produces $O(n^{n/2})$ variables and constraints and causes the solver to run out of memory on all but the smallest instances. In this experiment, we compare the new search-based NCP solver with and without learning with QeCode. It can be seen from Table 1 that our search-based NCP solver does significantly better than QeCode on this problem. Even without learning, we are much faster, due to the fact that we have variables representing the output values of subproblems and it is possible to propagate domain changes on them. Such variables do not exist in Qecode. Nogood learning provides a massive benefit due to its ability to explain successes. It is able to learn that if there is a winning strategy with a particular number of matches where you take k matches next, then in any other branch where you had the same number of matches but the limit on the number of matches you can take is greater than or equal to k, its a won game. In fact, the asymptotic complexity of the NCP solver is polynomial when nogood learning is used, as opposed to $O(n^n)$ if no learning is used.

The Network Links Pricing problem [3] is described in Example 6. The results are shown in Table 2. The no-learn solver is substantially faster than QeCode, which appears to be because Qecode does not use branch and bound during solving this problem. Nogood learning is only slightly beneficial for this problem, giving a constant factor reduction in node count and run times. This is not surprising, as the problem is very shallow (only 3 layers), and there is only a single inequality constraint in the final layer, so there is not much propagation going on, and thus not much opportunity for learning.

Table 2. Comparison of the search-based NCP solver without learning (no-learn) and with learning (learn) with QeCode (qecode) on the network link pricing problem.

Stages	no-learn		learn		qecode	
	fails	*time*	*fails*	*time*	*fails*	*time*
6	37399	2.63	5037	1.80	376234	7.10
7	342823	23.06	52301	18.54	2218653	44.48
8	985622	68.24	121815	56.82	12148442	255.66
9	17566514	1225.80	2253269	903.01	—	TO

Table 3. Comparison on a simple production planning problem of the search-based NCP with solver learning (learn) with published results (from [1], run on a different machine) for the search-based stochastic CSP solver of [2] (search) and the scenario-based solver of [1] (scen), also showing how learn scales to larger numbers.

Stages	no-learn		learn		search		scen	
	fails	*time*	*fails*	*time*	*fails*	*time*	*fails*	*time*
1	7	0.01	8	0.01	10	0.01	4	0.00
2	178	0.01	16	0.01	148	0.03	8	0.02
3	4574	0.36	24	0.01	3604	0.76	24	0.16
4	116305	9.24	32	0.01	95570	19.07	42	1.53
5	2955371	233.05	40	0.01	2616858	509.95	218	18.52
6	—	TO	48	0.01	—	TO	1260	474.47

Stages	learn	
	fails	*time*
10	80	0.03
20	160	0.11
30	240	0.27
40	320	0.53
50	400	0.93
100	800	7.40

Consider the simple production planning problem of Example 3 which was examined in [2] using search-based approaches and [1] using unrolling or scenario generation. From Table 3, it is clear there is an asymptotic difference in complexity. This occurs since each subproblem only involves a single input parameter, i.e., how much stock we currently have. If the stock is over the maximum demand for this period, then increasing it does not help at all, since we can just produce it during the next period instead. Nogood learning is able to derive a nogood that expresses this. Similarly, if the current stock is under the minimum demand for this period, then it fails in all scenarios, and again, nogood learning is able to derive a nogood that expresses this. Thus at each stage, the solver only has to try $O(S)$ values for the production, where S is the number of different possible demands, and we end up needing to examine only $O(Sk)$ nodes where k is the number of stages. If we do not use nogood learning, then our solver has virtually identical behaviour to the search based approach of [1] (search) and has an exponential complexity. Although our learning solver only requires a linear number of nodes, the run time appears to be growing as $O(k^3)$ due to the fact that as k increases, we have more variables and constraints to propagate at each node.

6 Related Work

NCPs are a very general form of optimization problem. They include stochastic CSP/COP, quantified CSP/COP, QCOP+ [3], QBF, bi-level and multi-level programming. The evaluation approach we define for NCP is a generalization of the search-based approaches used for many of these problems, although not

many focus on propagation, and only QBF solvers also consider learning. NCPs are more general than these other formalisms principally because aggregator terms can appear arbitrarily nested in both function terms and constraints.

Probably the closest work to NCPs is Quantified Constraint Optimization (QCOP+) [3]. QCOP+ are based on extending quantified CSPs to include local objective functions. They are limited compared to NCPs since only a single chain of nesting is allowed, meaning they cannot for example use the efficient form of the problem in Example 5. QCOP+ is implemented in a system QeCode which we compare with in the experiments section. They do not consider learning.

Another closely related work is the Plausibility-Feasibility-Utility (PFU) framework [16]. However, PFU does not allow tuple types to be the result of aggregator constraints, which means it cannot express bilevel problems such as the Network Link Pricing problem of Example 6. The tree search algorithms for evaluating PFUs is similar to that for NCPs, but they do not consider shortcircuit evaluation or learning. The PFU framework is studied theoretically in [16] and does not appear to have an implementation.

QBF is the form of NCP with only **and** and **or** aggregator constraints and clauses, and there is a significant body of work about how propagation and clause learning can be used in this context. The learning used in QBF is considerably simpler than for NCP, which tackles finite domain and interval variables and constraints and a much larger range of aggregators and more complicated nesting. SAT modulo theory solvers (e.g. Z3 [17]) are extended to handled quantified formula but principally through instantiation [18] akin to unrolling, which does not require (partial) search trees to be explained and only considers the **and** and **or** aggregators.

7 Conclusion

In summary, NCP's are a highly expressive formalism that unifies CSP/COP's, stochastic CSP/COP's, quantified CSP/COP's, bi-level and multi-level programming in the one framework, and allows many other kinds of nested problems to be expressed. We have demonstrated an effective search-based CP solver for evaluating them, which is significantly improved by the use of nogood learning to avoid repeating similar search. The resulting solver is competitive with or significantly better than state of the art CP-based approaches for many of the problems in these problem classes and brings us much closer to a universal CP solver that can "solve them all". Interesting directions for further investigation are: lazy or partial unrolling of aggregator constraints, model analysis and transformation for NCPs, improving propagation using the structure of the aggregation tree, approximation methods, and hybrid methods where each subproblem is solved using the technology most suited for it, e.g., using an LP or MIP propagator.

Acknowledgments

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. This work was partially supported by Asian Office of Aerospace Research and Development grant 12-4056.

References

1. Tarim, A., Manandhar, S., Walsh, T.: Stochastic Constraint Programming: A Scenario-Based Approach. Constraints **11**(1) (2006) 53–80
2. Walsh, T.: Stochastic Constraint Programming. In van Harmelen, F., ed.: ECAI, IOS Press (2002) 111–115
3. Benedetti, M., Lallouet, A., Vautard, J.: Quantified constraint optimization. In Stuckey, P.J., ed.: Principles and Practice of Constraint Programming, 14th International Conference. Volume 5202 of Lecture Notes in Computer Science., Springer (2008) 463–477
4. Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/term resolution and learning in the evaluation of quantified Boolean formulas. J. Artif. Intell. Res.(JAIR) **26** (2006) 371–416
5. Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean satisfiability solver. In: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, ACM (2002) 442–449
6. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3) (2009) 357–391
7. Feydy, T., Stuckey, P.J.: Lazy Clause Generation Reengineered. In Gent, I.P., ed.: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Volume 5732 of Lecture Notes in Computer Science., Springer (2009) 352–366
8. Frisch, A., Stuckey, P.: The proper treatment of undefinedness in constraint languages. In Gent, I., ed.: Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Volume 5732 of LNCS., Springer-Verlag (2009) 367–382
9. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints **13**(3) (2008) 229–267
10. Schwenk, A.: Take-away games. Fibonacci Quarterly **8** (1970) 225–234
11. Chen, H.: The Computational Complexity of Quantified Constraint Satisfaction. PhD thesis, Cornell University (2004)
12. Samulowitz, H.: Solving Quantified Boolean Formulas. PhD thesis, University of Toronto (2007)
13. Benedetti, M., Lallouet, A., Vautard, J.: Reusing CSP propagators for QCSPs. In: Recent Advances in Constraints. Springer (2007) 63–77
14. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. ACM Transactions on Programming Languages and Systems (TOPLAS) **31**(1) (2008) 2
15. Chu, G., Stuckey, P.: Inter-problem nogood learning in constraint programming. In Milano, M., ed.: Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming. Number 7514 in LNCS, Springer (2012) 238–247
16. Pralet, C., Verfailles, G., Schiex, T.: An algebraic graphical model for decision with uncertainties, feasibilities, and utilities. Journal of Artificial Intelligence Research **29** (2007) 421–489
17. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. (2008) 337–340
18. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In Pfenning, F., ed.: Proceedings of the Conference on Automated Deduction. Volume 4603 of Lecture Notes in Computer Science., Springer (2007) 167–182