Scenario-based Learning for Stochastic Combinatorial Optimisation

David Hemmi[12],
Guido Tack[12] and Mark Wallace[1]

[1] Faculty of IT, Monash University, Australia
[2] Data61/CSIRO, Australia
{david.hemmi, guido.tack, mark.wallace}@monash.edu

Abstract. Combinatorial optimisation problems often contain uncertainty that has to be taken into account to produce realistic solutions. This uncertainty is usually captured in *scenarios*, which describe different potential sets of problem parameters based on random distributions or historical data. While efficient algorithmic techniques exist for specific problem classes such as linear programs, there are very few approaches that can handle general Constraint Programming formulations with uncertainty. This paper presents a generic method for solving stochastic combinatorial optimisation problems by combining a *scenario-based decomposition* approach with *Lazy Clause Generation* and strong *scenario-independent nogoods* over the first stage variables. The algorithm can be implemented based on existing solving technology, is easy to parallelise, and is shown experimentally to scale well with the number of scenarios.

1 Introduction

Reasoning under uncertainty is important in combinatorial optimisation, since uncertainty is inherent to many problems in an industrial setting. An example of this especially hard class of optimisation problems is the generalised assignment problem [2], where jobs with an uncertain duration have to be assigned to computers in a network, or the assignment of tasks to entities when developing a project plan. Other examples include the stochastic Steiner tree problem with uncertainty in the set of terminals [12]. At first, edges in the graph can be purchased at a low price. Once the set of terminals is revealed, the Steiner tree must be completed by purchasing edges at an increased price. Various forms of the stochastic set covering problem [11] and stochastic vehicle routing problems [21] are of similar form.

The focus of this paper is on stochastic optimisation problems with a combinatorial structure. The random variables describing the uncertainty have finite support. Each random variable has an underlying discrete probability distribution. A scenario describes the stochastic problem when all the random variables are fixed. Each scenario has a probability of occurrence. Stochastic problems are composed of a first and subsequent stages, the simplest being a two stage problem. The first stage denotes the problem before information about the random variables is revealed and first stage decisions are taken with respect to all scenarios. Second stage decisions are made once the random variables are fixed.

In constraint programming (CP), modeling frameworks that allow solver agnostic modelling have been developed. Examples include MiniZinc [15] and Essence [10]. Problems described in one of these languages can be solved with a range of CP, MIP or SAT solvers without the user having to tailor the model for specific solvers. Stochastic MiniZinc [17] is an extension of the MiniZinc modelling language that supports uncertainty. Stochastic MiniZinc enables modellers to express combinatorial stochastic problems at a high level of abstraction, independent of the stochastic solving approach. Stochastic MiniZinc uses the standard language set of MiniZinc and admits models augmented with annotations describing the first and second stage. Stochastic MiniZinc automatically transforms the model into a structure that can be solved by standard MiniZinc solvers. At present, stochastic MiniZinc translates a two-stage model into the deterministic equivalent and uses a standard CP, MIP or SAT solver for the search. However, standard solver technology cannot exploit the special model structure of scenario based stochastic optimisation problems. As a result, the search performance is poor and it is desirable to develop a solver that can handle this class of problems without requiring the modeler to have expert knowledge in stochastic optimisation.

The literature on stochastic optimisation problems containing integer variables is diverse, see ([1, 4, 6, 13, 18]). Previous works have been concerned with problems that have a specific mathematical structure, for example linear models with integrality requirements. Furthermore, these methods are not available as standalone solvers and it is not possible to easily transform a model defined in a CP framework into a form that can be solved by one of these algorithms, in particular when a CP model contains non-linear constraints that would require (potentially inefficient) linearisation. In addition, methods to solve stochastic constraint satisfaction problems, inspired by the stochastic satisfiability problem, have been introduced, see ([3, 14]) for an overview.

This paper proposes an algorithm to solve combinatorial stochastic optimisation problems. The work is based on Lazy Clause Generation (LGC) and translates the *scenario decomposition algorithm for 0-1 stochastic programs* introduced by Ahmed [1] into the CP setting. Our main contributions are the following: first, a search strategy that in combination with Ahmed's algorithm produces scenario-independent nogoods; secondly, an effective use of nogoods in the sub-problems when using decomposition algorithms; thirdly, a scenario bundling method to further improve the performance. Also, the benchmark instances we are using have been made publicly available on the CSPLib.

In contrast to the methods in the literature, the proposed algorithm can be used directly as a back-end solver to address problems formulated in a CP framework. No sophisticated reformulation of the model nor problem specific decompositions are required. The paper concludes with an experiment section.

2 Background and Related Work

This section introduces the basic notation, and then discusses relevant literature.

2.1 Basic definitions

We will base our definition of stochastic problems on the deterministic case.

Definition 1 *A (deterministic)* ***constraint optimisation problem*** *(COP) is a four-tuple P defined as follows:*

$$P = \; < V, D, C, f >$$

where V is a set of decision variables, D is a function mapping each element of V to a domain of potential values, and C is a set of constraints. A constraint $c \in C$ acts on variables $x_i, ..., x_j$, termed $scope(c)$ and specifies mutually-compatible variable assignments σ from the Cartesian product $D(x_i) \times ... \times D(x_j)$. The quality of a solution is measured using the objective function f. We write $\sigma(x)$ for the value of x in assignment σ; $\sigma|_X$ for σ restricted to the set of variables X; and $\sigma \in D$ to mean $\forall x : \sigma(x) \in D(x)$. We define the set of ***solutions*** *of a COP as the set of assignments to decision variables from the domain D that satisfies all constraints in C:*

$$sol(P) = \{\sigma \in D \mid \forall c \in C : \sigma|_{scope(c)} \in c\}$$

Finally, an ***optimal solution*** *is one that minimises the objective function:*

$$\min_{\sigma \in sol(P)} f(\sigma)$$

On the basis of a COP we can define a Stochastic Constraint Optimisation Problem (SCOP). We restrict ourselves to two-stage problems, although the definitions and algorithms can be generalised to multi-stage problems. In a two-stage SCOP, a set of decisions, the *first stage* decisions, is taken before the values for the random variables are known. Once the random variables are fixed, further decisions are made for the subsequent stages. To enable reasoning before the random variables are revealed, we assume that their values can be characterised by a finite set of *scenarios*, i.e., concrete instantiations of the random variables based on historical data or forecasting. Our task is then to make first stage decisions that are optimal *on average* over all scenarios.

The idea behind our definition of an SCOP is to regard each scenario as an individual COP (fixing all random variables), with the additional restriction that all scenario COPs share a common set of first stage variables V. The goal is to find solutions to the scenarios that agree on the first stage variables, and for which the weighted average over all scenario objective values is minimal.

Definition 2 *A* ***stochastic constraint optimisation problem*** *(SCOP) is a tuple as follows:*

$$\begin{aligned} P \quad &= < V, P_1, \ldots, P_k, p_1, \ldots, p_k > \\ \textit{with } P_i &= < V_i, D_i, C_i, f_i > \qquad\qquad \forall i \in 1..k : V \subseteq V_i \end{aligned}$$

where each P_i is a COP, each p_i is the weight (e.g. the probability) of scenario i, and the set V is the set of first stage variables shared by all P_i.

The set of ***solutions*** *of an SCOP is defined as tuples of solutions of the P_i that agree on the variables in V:*

$$\begin{aligned} sol(P) = \{\sigma | \sigma =< \sigma_i, .., \sigma_k >, \ & \sigma_i \in sol(P_i), \\ & \sigma_i(x) = \sigma_j(x) \\ & \forall i, j \in 1..k, x \in V\} \end{aligned}$$

An ***optimal solution*** *to an SCOP minimises the weighted sum of the individual objectives:*

$$\min_{\sigma \in sol(P)} \textstyle\sum_{i=1}^{k} p_i f_i(\sigma_i)$$

Multiple methods to solve stochastic programs have been developed in the past. The most direct option is to formulate the SCOP as one large (deterministic) COP, called the *deterministic equivalent* (DE). Starting from Definition 2, the DE is constructed simply by taking the union of all the P_i (assuming an appropriate renaming of the second stage variables), and using the weighted sum as the objective function. The number of variables and constraints in the DE increases linearly with the number of scenarios. However, in case of a combinatorial second stage, the solving time may increase exponentially. As a result the DE approach lacks scalability.

An alternative to the DE is to *decompose* the SCOP, in one of two ways. Firstly, the problem can be relaxed by time stages. In the *vertical* decomposition a master problem describes the first stage and contains an approximation of the second stage. A set of sub-problems captures the second stages for each scenario. Complete assignments to the master problem are evaluated against the sub-problems. The L-shaped method [4], which is similar to Benders decomposition, is an example for an algorithm that works on the basis of the *vertical* decomposition. However, the L-Shaped method is limited to linear, continuous SCOPs, with extensions available for problems with integrality requirements. Secondly, the problem can be decomposed *horizontally*, by scenarios. The rest of this paper is based on this scenario decomposition, which is introduced in the next section.

2.2 Scenario-based decomposition

A straightforward decomposition for SCOPs is the scenario decomposition, where the P_i of an SCOP are treated as individual problems. A feasible solution to the stochastic program requires the shared variables to agree across all scenarios.

To ensure feasibility, an additional *consistency constraint* $\sigma_i(x) = \sigma_j(x) \forall i, j \in 1 \ldots k, x \in V$ is required. Note how this constraint was built into our definition of SCOP, but now has become an external requirement that can be relaxed. To make this more formal, we introduce the notion of a *pre-solution*, which represents tuples of individual scenario solutions that may violate the consistency constraint.

Definition 3 *A* ***pre-solution*** *to a SCOP P is a tuple*

$$pre\_sol(P) = \{\sigma | \sigma =< \sigma_i, .., \sigma_k >, \sigma_i \in sol(P_i), \forall i \in 1..k\}$$

We can redefine the set of solutions as exactly those pre-solutions that agree on the first stage variables:

$$sol(P) = \{\sigma | \sigma \in pre\_sol(P) \ \wedge \sigma_i(x) = \sigma_j(x) \ \forall i, j \in 1 \ldots k, x \in V\}$$

Algorithms based on the scenario decomposition generate pre-solutions and iteratively enforce convergence (i.e., consistency) on the first stage variables V, using different methods as described below. Any converged pre-solution is a feasible solution of the SCOP, and yields an upper bound on the stochastic objective value. As pre-solutions relax the consistency constraints, any pre-solution that minimises the individual objective functions represents a lower bound to the stochastic objective:

$$\min_{\sigma \in pre\_sol(P)} \sum_{i=1}^{k} p_i f_i(\sigma_i) = \sum_{i=1}^{k} p_i \min_{\sigma \in sol(P_i)} f_i(\sigma) \leq \min_{\sigma \in sol(P)} \sum_{i=1}^{k} p_i f_i(\sigma_i)$$

A number of algorithms have been developed on the basis of iteratively solving the scenarios. In the following, we will introduce the most relevant methods. Progressive Hedging (PH) finds convergence over the shared variables by penalising the scenario objective functions. The added penalty terms represent the Euclidean distance between the averaged first stage variables and each scenario. Gradually, by repetitively solving the scenarios and updating the penalty terms, convergence over the shared variables is achieved. Originally PH was introduced by Rockafellar and Wets [18] for convex stochastic programs, for which convergence on the optimal solution can be guaranteed. However, PH was successfully used as a heuristic for stochastic MIPs, where optimality guarantees do not hold due to non-convexity, such as the stochastic inventory routing problem in [13] or resource allocation problems in [22] to name just a few.

An alternative scenario-based decomposition method was proposed by Carø and Schultz [6], called *Dual Decomposition* (DD), a branch and bound algorithm for stochastic MIPs. To generate lower bounds, the DD uses the dual of the stochastic problem, obtained by relaxing the consistency constraints using Lagrangian multipliers. To generate upper bounds, the solutions to the subproblems are averaged and made feasible using a rounding heuristic. The feasible region is successively partitioned by branching over the shared variables. The efficiency of the DD strongly depends on the update scheme used for the Lagrangian multipliers.

The algorithms introduced so far, including PH, DD and the L-Shaped method, only work for linear programs, with some extensions being available for certain classes of integer linear programs. As a result these approaches may require a solid understanding of the problem and the algorithm, in order to reformulate a given problem to be solved or to adapt the algorithm.

2.3 Evaluate and Cut

In contrast to the specialised approaches, Ahmed [1] proposes a scenario decomposition algorithm for SCOPs with binary first stage variables, which does not rely on the constraints being linear or the problem convex. Ahmed's algorithm solves each scenario COP independently to optimality. This yields a pre-solution to the SCOP and according to the discussion above a lower bound. In addition, each scenario first stage solution is evaluated against all other scenarios. The result is a candidate solution to the SCOP and therefore an upper bound. The evaluated candidates are added to the scenarios as *nogoods* (or *cuts*), forcing them to return different solutions in every iteration. As long as the first stage variables have finite domains, this process is guaranteed to find the optimal SCOP solution and terminates, either when the lower bound exceeds the upper bound or no additional candidate solutions are found. The scenarios can be evaluated separately, allowing highly parallelised implementations.

Algorithm 1 implements Ahmed's [1] ideas using our SCOP notation. We call this algorithm EVALUATEANDCUT, as it evaluates each candidate against all scenarios and adds nogoods that cut the candidates from the rest of the search. In line 7, each scenario i is solved independently, resulting in a variable assignment to the first and second stage σ, and a value *obj* of the objective function f_i. The lower bound is computed incrementally as the weighted sum (line 8). Furthermore, a set S of all first stage scenario solutions is constructed(line 9, recall that $\sigma|_V$ means the restriction of σ to the first stage variables V). Each candidate solution $\sigma|_V$ in S is evaluated against all scenarios P_i (line14). This yields a feasible solution to the SCOP, and the tentative upper bound t_{UB} is updated. Once the candidate $\sigma|_V$ is evaluated, a *nogood* excluding $\sigma|_V$ from the search is added to the scenario problems P_i (line 16). We call this the **candidate nogood**. If the tentative upper bound is better than the global upper bound UB, a new incumbent is found (lines 17–19).

Each candidate $\sigma|_V$ is an optimal solution to one scenario problem P_i; excluding it from all scenarios in line 16 implies a monotonically increasing lower bound. The algorithm terminates when the lower bound exceeds the upper bound. Note, the call to solve in line 7 will return infinity if any scenario becomes infeasible. Line 21 will be used in our extensions of the algorithm presented in Sect. 3 and can be ignored for now.

Ahmed [1] reports on an implementation of this algorithm using MIP technology, and evaluates it on benchmarks with a binary first and linear second stage. The paper claims that this algorithm can find high quality upper bounds quickly. For linear SCOPs, improvements to the algorithm are introduced in

[20], taking advantage of a linear relaxation to improve the lower bounds, and additional optimality cuts based on the dual of the second stage problem.

Algorithm 1 A scenario decomposition algorithm for 0-1 stochastic programs

```
1: procedure EVALUATEANDCUT
2:     Initialise: UB = $\infty$, LB = -$\infty$, sol = NULL
3:     while LB < UB do
4:         LB = 0, S= $\varnothing$
5:         % Obtain lower bound and find candidate solutions
6:         for i in 1..k do
7:             < $\sigma$,obj> = SOLVE(P$_i$)
8:             LB += p$_i$ * obj
9:             S $\cup$ = $\sigma|_V$
10:        % Check candidate solutions and obtain upper bound
11:        for $\sigma_V \in$ S do
12:            t$_{UB}$ = 0
13:            for i in 1..k do
14:                <_,obj> = solve(P$_i$[C $\cup$ = {$\sigma_V$}])
15:                t$_{UB}$ += p$_i$ * obj
16:                P$_k$ = P$_k$[C $\cup$ = {$\neg\sigma_V$}]
17:            if t$_{UB}$ < UB then
18:                sol = $\sigma_V$
19:                UB = t$_{UB}$
20:        % Evaluate partial first stage assignments
21:        DIVE(P,UB,S,sol)
22:    return sol
```

Discussion: The EVALUATEANDCUT algorithm has the distinct advantage over other decomposition approaches that it can be applied to arbitrary SCOPs. This makes it an ideal candidate as a backend for solver-independent stochastic modelling languages such as Stochastic MiniZinc. However, in the worst case the algorithm requires $O(k^2)$ checks for k scenarios in each iteration. This is not prohibitive if evaluating the candidates is cheap. However, for problems that have many scenarios and a combinatorial second stage, this quadratic behaviour may dominate the solving time. In the following section, we propose a technique to decrease the time to solve and evaluate the individual candidates. Furthermore, we propose a method to reduce the number of iterations required to solve the SCOP.

3 Scenario-based Algorithms for CP

This section introduces three modifications to EVALUATEANDCUT that improve its performance: using *Lazy Clause Generation* solvers for the scenario sub-problem; using *dives* to limit the number of candidate verifications and iterations required; and *scenario bundling* as a hybrid between the DE and scenario-based decomposition.

Traditional CP solvers use a combination of propagation and search. Propagators reduce the variable domains until no further reduction is possible or a constraint is violated. Backtracking search, usually based on variable and value selection heuristics, explores the options left after propagation has finished. In contrast to traditional CP solvers, Lazy Clause Generation (LCG) [16] solvers *learn* during the search. Every time a constraint is violated the LCG solver analyses the cause of failure and adds a constraint to the model that prevents the same failure from happening during the rest of the search. The added constraints are called *nogoods*, and they can be seen as the CP equivalent of cutting planes in integer linear programming – they narrow the feasible search space.

Chu and Stuckey showed how nogoods can be reused across multiple instances of the same model if the instances are structurally similar [7]. This technique is called *inter-instance learning*. The nogoods learned during the solving of one instance can substantially prune the search space of future instances. Empirical results published in [7] indicate that for certain problem classes, a high similarity between models yields a high resusability of nogoods and therefore increased performance.

The evaluateAndCut algorithm repeatedly solves very similar instances. In each iteration, every scenario is solved again, with the only difference being the added nogoods or the projection onto the first stage variables to evaluate candidates (see line 16 in Algorithm 1). As a consequence, inter-instance learning can be applied to speed up the search within the same scenario. We call this concept *vertical learning*. No changes are required to Algorithm 1, except that we assume the calls to solve in line 14 to be incremental and remember the nogoods learned for each P_i in the previous iteration. To the best of our knowledge, this is the first use of inter-instance learning in a vertical fashion.

3.1 Search Over Partial Assignments

As introduced earlier, the evaluateAndCut algorithm quickly finds high quality solutions, by checking the currently best solution for an individual scenario (the *candidate*) against all other scenarios. However, in order to prove optimality, evaluateAndCut relies on the lower bound computed from the pre-solution found in each iteration. The quality of the lower bound, and the number of iterations required for it to reach the upper bound and thus prove optimality, crucially depends on the candidate nogoods added in each iteration.

Compared to nogoods as computed during LCG search, candidate nogoods are rather weak: They only cut off a single, complete first stage assignment. Furthermore, in the absence of Lagrangian relaxation or similar methods, candidate nogoods are the *only* information about the global lower bound that is available to each scenario problem P_i.

Stronger nogoods: To illustrate the candidate nogoods produced by evaluateAndCut, consider a set of shared variables of size 5, and a first stage candidate solution $x = [3, 6, 1, 8, 3]$. The resulting candidate nogood added to the constraint set of each scenario subproblem P_i is:

$$P_i = P_i[C \cup = \{x_1 \neq 3 \vee x_2 \neq 6 \vee x_3 \neq 1 \vee x_4 \neq 8 \vee x_5 \neq 3\}]$$

The added constraint cuts off exactly one solution. A nogood composed of only a *subset* of shared variables would be much stronger. For example, assume that we can prove that even the *partial* assignment $x_1 = 3 \wedge x_2 = 6 \wedge x_3 = 1$ cannot be completed to an optimal solution to the stochastic problem. We could add the following nogood:

$$P_i = P_i[C \cup = \{x_1 \neq 3 \vee x_2 \neq 6 \vee x_3 \neq 1\}]$$

In contrast to the original candidate nogood that pruned exactly one solution, the new, shorter nogood can cut a much larger part of the search space. We call this stronger kind of nogood a **partial candidate nogood**.

Diving: We now develop a method for finding partial candidate nogoods based on *diving*. The main idea is to iteratively fix first stage variables across all scenarios and compute a pre-solution, until the lower bound exceeds the global upper bound. In that case, the fixed variables can be added as a partial nogood.

The modification to Algorithm 1 consists of a single added call to a procedure DIVE in line 21, which is executed in each iteration after the candidates have been checked. The definition of DIVE is described in Algorithm 2.

Line 5 constructs a constraint c that fixes a subset of the first stage variables, based on the current set of scenario solutions S. This is done according to a heuristic that is introduced later. The loop in lines 7–10 is very similar to the computation of a pre-solution in Algorithm 1, except that *all* scenarios are forced to agree on the selected subset of shared variables by adding the constraint c. As in Algorithm 1, we compute a lower bound based on the pre-solution. However, since we have arbitrarily forced the scenario sub-problems to agree using c, this lower bound is not valid globally – there can still be better overall solutions with different values for the variables $\sigma|_V$.

In essence, the algorithm is adding specific consistency constraints one by one. Adding such constraint can lead to one of three states:

1. The lower bound does not exceed the upper bound, but all scenarios agree on a common first stage assignment (even if c does not constrain all first stage variables). This means that a new incumbent solution is found (lines 11–17) and the constraint c can be added as a partial candidate nogood.
2. The lower bound meets or exceeds the upper bound. In this case, the partial consistency constraint c cannot be extended to any global solution that is better than the incumbent (lines 18–22). We can therefore add c as a partial candidate nogood.
3. The lower bound is smaller than the upper bound, and the scenarios have not converged on the first stage variables. In this case, an additional constraint is added to the partial consistency constraint c.

Algorithm 2 Searching for partial candidate nogoods using diving

```
 1: procedure DIVE(P,UB,S,sol)
 2:     $t_{LB}$ = -$\infty$
 3:     while $t_{LB}$ < UB do
 4:         $t_{LB}$ = 0
 5:         $c$ = SELECTFIXED(S,V) % Select first stage variables to fix
 6:         S= $\varnothing$
 7:         for i in 1..k do
 8:             <$\sigma$,obj> = SOLVE($P_i$[C $\cup$= {$c$}])
 9:             $t_{LB}$ += $p_i$ * obj
10:             S $\cup$= $\sigma|_V$
11:         if $t_{LB}$ < UB $\wedge$ $S = \{\sigma_V\}$ then
12:             UB = $t_{LB}$
13:             sol = $\sigma_V$
14:             % Add partial candidate nogood
15:             for i in 1..k do
16:                 $P_i$ = $P_i$[C $\cup$= {$\neg c$}]
17:             return
18:         if $t_{LB}$ >= UB then
19:             % Add partial candidate nogood
20:             for i in 1..k do
21:                 $P_i$ = $P_i$[C $\cup$= {$\neg c$}]
22:             return
```

The DIVE procedure terminates because in each iteration, SELECTFIXED fixes at least one additional first stage variable, which means that either case (1) or (2) above must eventually hold.

Note, we do not evaluate the scenario solutions constructed during diving against all scenarios. The rationale is that the additional consistency constraints are likely to yield non-optimal overall solutions (i.e., worse than the current incumbent). The $O(k^2)$ evaluations would therefore be mostly useless.

Diving heuristic: Let us now define a heuristic for choosing the consistency constraints to be added in each iteration during a dive. The goal is to produce short, relevant partial candidate nogoods and achieve convergence across all scenarios quickly. Our heuristic focuses on the first stage variables that have already converged in the current pre-solution. The procedure SELECTANDFIX in Algorithm 3 implements the heuristic. First, it picks all the first stage variables that have already converged, including those fixed in previous steps of this dive (line 5 right side of $\wedge$). Secondly, an additional first stage variable assignment is chosen (line 5 left side of $\wedge$), based on the variable/value combination that occurs most often over all scenarios. Given the current pre-solution S, it first constructs a mapping *Count* from variables to multisets of their assignments (line 3). Thereafter a variable/value combination is picked that occurs most often, but is not converged yet (line 4). In example, if variable x_3 is assigned to the value 4 in three scenario solutions, and to the value 7 in another two, then *Count* would contain the pair $< x_3, \{4, 4, 4, 7, 7\} >$. The value 4 would be assigned to x_3 assuming it is the most prevalent variable/value combination over all first stage variables. Finally, we construct a constraint that assigns x_e to v_e, in addition to assigning all variables that have converged across all scenarios (line 5).

Algorithm 3 Diving heuristic

```
1: procedure SELECTFIXED(S,V)
2:     $Vals =< \{\sigma_i(x) : \sigma_i \in S\} : x \in V >$
3:     $Count =< card(\{i : \sigma_i(x) = val, \sigma_i \in S\}) :< x \in V, val \in Vals_x >>$
4:     $< x_e, v_e > = \underset{\substack{<x,v> \\ val \in Vals_x \\ Count_{<x,v>} < k}}{\arg\max} Count_{<x,val>}$
5:     $c = (x_e = v_e \wedge \bigwedge_{\substack{x \in V \\ val \in Vals_x \\ Cout_{<x,v>} = k}} x = v)$
6:     return $c$
```

Scenario Bundling: The final extension of Ahmed's algorithm is based on the observation that the sub-problems in any scenario decomposition method can in fact comprise multiple scenarios, as long as we can find the optimal *stochastic* solution for that subset of scenarios in each iteration. We call this *scenario bundling*.

Since EVALUATEANDCUT has $O(k^2)$ behaviour for k scenarios, bundling can have a positive effect on the runtime, as long as the solving time for each bundle is not significantly higher than that for an individual scenario. Furthermore, the bundling of scenarios yields better lower bounds, since each component of a pre-solution is now an optimal stochastic solution for a subset of scenarios, which is guaranteed to be worse than the individual scenario objectives. By bundling scenarios in this way, we can therefore combine the fast convergence of EVALUATEANDCUT for large numbers of scenarios with the good performance of methods such as the DE on low numbers of scenarios. Scenario bundling has been applied to progressive hedging [9] and to EVALUATEANDCUT in [19].

4 Experiments

This section reports on our empirical evaluation of the algorithms discussed above. As a benchmark set we use a stochastic assignment problem with recourse similar to the stochastic generalised assignment problem (SGAP) described in [2]. A set of jobs, each composed of multiple tasks, is to be scheduled on a set of machines. Precedence constraints ensure that the tasks in a job are executed sequentially. Furthermore, tasks may be restricted to a sub-set of machines.

The processing time of the tasks varies across the set of machines, and in the stochastic version of the problem, this is a random variable. In the first stage, tasks must be assigned to machines. An optimal schedule with respect to the random variables is created in the second stage. The objective is to find a task to machine assignment minimizing the expected makespan over all scenarios.

Work on the SGAP with uncertainty on whether a job must be executed is described in [2]. However, to the best of our knowledge, there are no public benchmarks for the SGAP with uncertain processing times. Benchmarks for our experiments are created as described for deterministic flexible job shop instances in [5]. The scenarios are created by multiplying the base task durations by a number drawn from a uniform distribution with mean 1.5, variance 0.3, a lower limit of 0.9 and upper limit of 2.

We modelled the benchmark problems in MiniZinc. Each scenario is described in a separate MiniZinc data file, and compiled separately. This enables the MiniZinc compiler to optimise the COPs individually before solving. The models use a fixed search strategy (preliminary studies using activity based search did not improve the performance). The solver is implemented using Chuffed [8] and Python 2.7. The scenarios are solved using Chuffed, and learned nogoods are kept for subsequent iterations to implement vertical learning. A Python script coordinates the scenarios and dives. Up to twenty scenarios are solved in parallel. The experiments were carried out on a 2.9 GHz Intel Core i5, Desktop with 8 GB running OSX 10.12.1. A timeout of 3600 seconds was used.

4.1 Results

Table 1 contains the results for 9 representative problem instances with a range of 20 to 400 scenarios. In every instance, the optimal solution is found within the first few iterations and the remaining time is used to prove optimality. No results for the deterministic equivalent are presented as the run time is not competitive once the number of scenarios exceeds 20.

The impact of diving: The first two rows per instance contain the time it takes to solve the problem instances without scenario bundling. No substantial time difference can be reported for finding the optimal solution when dives are enabled. However, using dives improves the overall search performance in every instance. Figure 1 contains plots displaying the impact of dives when solving 100 scenarios without bundling.

The monotonically increasing graphs are the lower bounds. The horizontal black line is the optimal solution. The upper bound progress is not displayed, as the optimal solution is always found within a few seconds. Once the lower bound meets the upper bound, optimality is proven and the search terminates. The lower bound increases quickly at the beginning of the search. Over time the *evaluate and cut* method without diving flattens and the lower bound converges slowly towards upper bound. In strong contrast is the progress of the lower bound when using diving. At first during the initial iterations, the partial candidate nogoods are not showing any effects and the two curves are similar.

However, after the initial phase, generated strong partial candidate nogoods are paying off and the lower bound jumps drastically.

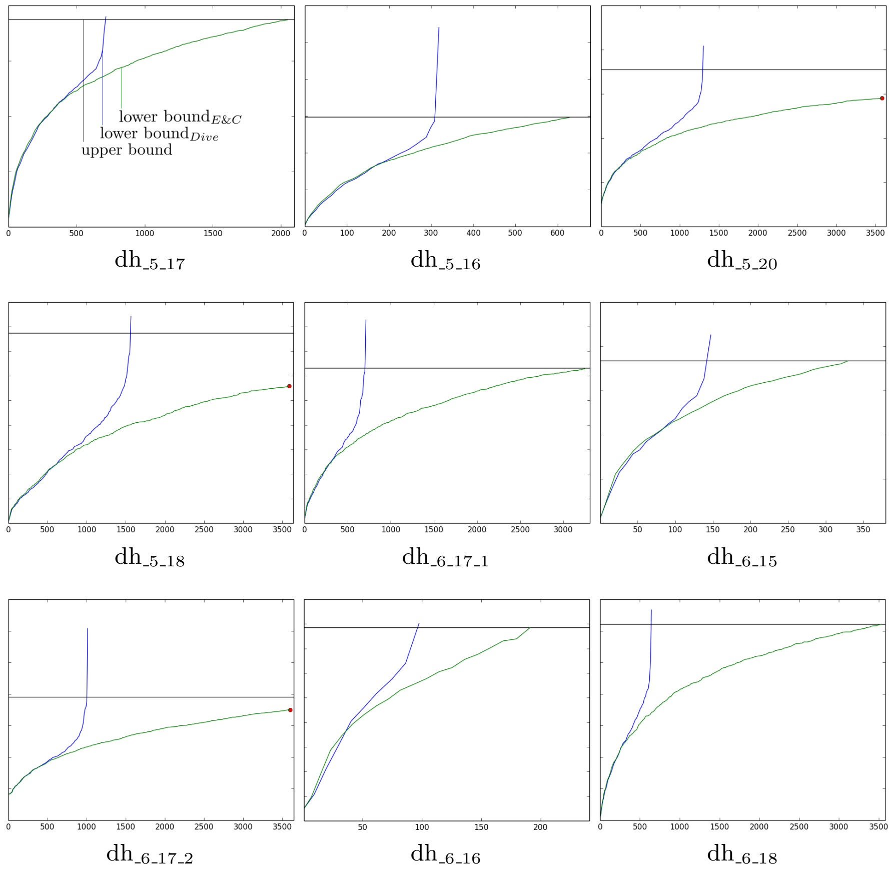

Fig. 1: The impact of dives

Using DE to proof optimality: The third row in each instance displays the times it takes to prove optimality using the deterministic equivalent. The upper bounds obtained from evaluateAndCut are used to constrain the stochastic objective function in the DE. Similarly to using the DE directly, it does not scale with an increased number of scenarios. Furthermore, in contrast to evaluate-AndCut no optimality gap is produced.

Scenario bundling: The last two rows per instance contain the time it takes to solve the instances using scenario bundling. Four scenarios are randomly grouped to form a DE. Each scenario group becomes a sub-problem solved with EVALUATEANDCUT with and without diving enabled. As a result, the run time decreases in every instance. For the benchmark instances, diving is less powerful when using scenario bundles. This can be explained by the decreasing number of iterations required to find the optimal solution. More time is spend solving the sub-problems and less effort is required to coordinate the scenarios.

Vertical learning: Table 2 shows the speed-up when using vertical learning. Each column displays the speed-up over a scenario group. Overall the solving time decreased by 19.3% with 10.7% variance with an increase of at most 72%.

		20	20	40	40	80	80	100	100	200	300	400
E&C	mean	23.4	29.6	15.8	19.2	9.7	27.4	7.2	26.0	21.8	19.6	17.6
	variance	6.0	17.2	5.0	37.1	2.4	13.8	3.3	12.0	10.7	10.6	8.4
Dive	mean	21.4	32.7	13.7	28.7	8.1	29.9	5.9	23.8	17.1	14.7	12.4
	variance	6.3	23.6	4.0	21.9	2.4	16.6	2.1	11.5	8.9	6.8	5.8

Speedup: 100 - 100 / time no learning * time with learning
Colored: Speedup in [%] when using vertical learning without bundling scenarios
White: Speedup in [%] when using vertical learning and bundling scenarios

Fig. 2: Speedup using vertical learning

5 Conclusion

This paper has presented the first application of Ahmed's scenario decomposition algorithm [1] in a CP setting. Furthermore, we have introduced multiple algorithmic innovations that substantially improve the EVALUATEANDCUT algorithm. The most significant improvement is the partial search to create strong candidate nogoods. All our algorithmic innovations can be implemented in a parallel framework.

To further strengthen the EVALUATEANDCUT algorithm we will continue to work on the following ideas. First, the heuristic used to determine the variables to be fixed strongly impacts the performance of the search. A good heuristic is able to produce strong, relevant nogoods. Introducing a master problem that enables a tree search and improved coordination is worthwhile exploring. For the results we have used the standard system to manage nogoods within Chuffed. Further analysing to role of nogoods will help us understand their impact and how we can use nogoods to improve vertical learning and efficiently incorporate inter-instance learning.

Instance	Algorithm	20	40	80	100	200	300	400
dh_5_17	E&C	24	343	1748	2060			
	Dive	12	152	584	716			
	DE$_{upperBound}$	7	215	1342	1923	-	-	-
	E&C$_{bundle}$	1	**8**	**19**	17	27	**49**	**46**
	Dive$_{bundle}$	1	11	20	17	**26**	45	64
dh_5_16	E&C	36	111	436	638			
	Dive	23	56	212	318			
	DE$_{upperBound}$	20	131	877	1136	-	-	-
	E&C$_{bundle}$	**2**	**5**	**9**	**11**	**34**	**69**	**91**
	Dive$_{bundle}$	3	7	12	13	41	72	93
dh_5_20	E&C	338	196	-	-			
	Dive	87	880	713	1304			
	DE$_{upperBound}$	43	293	2355	-	-	-	-
	E&C$_{bundle}$	**3**	**17**	**36**	72	240	423	645
	Dive$_{bundle}$	7	18	43	**71**	**190**	**328**	**510**
dh_5_18	E&C	163	719	-	-			
	Dive	50	179	984	1566			
	DE$_{upperBound}$	**3**	50	1547	3467	-	-	-
	E&C$_{bundle}$	7	**31**	162	226	407	712	1233
	Dive$_{bundle}$	8	32	**157**	**218**	**401**	**629**	**873**
dh_6_17_1	E&C	94	735	2328	3242			
	Dive	33	152	447	712			
	DE$_{upperBound}$	6	62	647	1073	-	-	-
	E&C$_{bundle}$	2	3	15	15	59	91	156
	Dive$_{bundle}$	2	3	**14**	15	**52**	**75**	**114**
dh_6_15	E&C	10	35	209	338			
	Dive	5	14	95	147			
	DE$_{upperBound}$	5	69	495	1142	-	-	-
	E&C$_{bundle}$	1	**3**	**16**	**21**	67	81	**138**
	Dive$_{bundle}$	1	4	19	25	**61**	**66**	183
dh_6_17_2	E&C	253	755	3345	-			
	Dive	58	153	725	1014			
	DE$_{upperBound}$	**0**	31	477	386	2159	-	-
	E&C$_{bundle}$	7	8	33	49	89	182	303
	Dive$_{bundle}$	6	**7**	**24**	**24**	**48**	**121**	**178**
dh_6_16	E&C	21	69	141	214			
	Dive	13	36	71	97			
	DE$_{upperBound}$	12	111	777	1921	-	-	-
	E&C$_{bundle}$	11	**24**	**29**	**42**	125	206	336
	Dive$_{bundle}$	11	25	37	54	**76**	**122**	**199**
dh_6_18	E&C	134	672	2140	3548			
	Dive	44	150	454	645			
	DE$_{upperBound}$	9	157	1790	3323	-	-	-
	E&C$_{bundle}$	**7**	**11**	**19**	**29**	**120**	219	338
	Dive$_{bundle}$	9	20	24	31	131	**212**	**310**

Table 1: Time to prove optimality [sec]

References

1. Ahmed, S.: A scenario decomposition algorithm for 0–1 stochastic programs. Operations Research Letters 41(6), 565–569 (2013)
2. Albareda-Sambola, M., Van Der Vlerk, M.H., Fernández, E.: Exact solutions to a class of stochastic generalized assignment problems. European journal of operational research 173(2), 465–487 (2006)
3. Balafoutis, T., Stergiou, K.: Algorithms for stochastic csps. In: CP. pp. 44–58. Springer (2006)
4. Birge, J.R., Louveaux, F.: Introduction to stochastic programming. Springer Science & Business Media (2011)
5. Brandimarte, P.: Routing and scheduling in a flexible job shop by tabu search. Annals of Operations research 41(3), 157–183 (1993)
6. CarøE, C.C., Schultz, R.: Dual decomposition in stochastic integer programming. Operations Research Letters 24(1), 37–45 (1999)
7. Chu, G., Stuckey, P.J.: Inter-instance nogood learning in constraint programming. In: CP. pp. 238–247. Springer (2012)
8. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne (2011)
9. Crainic, T.G., Hewitt, M., Rei, W.: Scenario grouping in a progressive hedging-based meta-heuristic for stochastic network design. Computers & Operations Research 43, 90–99 (2014)
10. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
11. Goemans, M., Vondrák, J.: Stochastic covering and adaptivity. In: Latin American symposium on theoretical informatics. pp. 532–543. Springer (2006)
12. Hokama, P., San Felice, M.C., Bracht, E.C., Usberti, F.L.: A heuristic approach for the stochastic steiner tree problem (2014)
13. Hvattum, L.M., Løkketangen, A.: Using scenario trees and progressive hedging for stochastic inventory routing problems. Journal of Heuristics 15(6), 527–557 (2009)
14. Manandhar, S., Tarim, A., Walsh, T.: Scenario-based stochastic constraint programming. arXiv preprint arXiv:0905.3763 (2009)
15. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: CP, pp. 529–543. Springer (2007)
16. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
17. Rendl, A., Tack, G., Stuckey, P.J.: Stochastic minizinc. In: CP. pp. 636–645. Springer (2014)
18. Rockafellar, R.T., Wets, R.J.B.: Scenarios and policy aggregation in optimization under uncertainty. Mathematics of operations research 16(1), 119–147 (1991)
19. Ryan, K., Ahmed, S., Dey, S.S., Rajan, D.: Optimization driven scenario grouping (2016)
20. Ryan, K., Rajan, D., Ahmed, S.: Scenario decomposition for 0-1 stochastic programs: Improvements and asynchronous implementation. In: Parallel and Distributed Processing Symposium Workshops. pp. 722–729. IEEE (2016)
21. Toth, P., Vigo, D.: Vehicle routing: problems, methods, and applications, vol. 18. Siam (2014)
22. Watson, J.P., Woodruff, D.L.: Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. Computational Management Science 8(4), 355–370 (2011)