The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

A Recursive Scenario Decomposition Algorithm for Combinatorial Multistage Stochastic Optimisation Problems

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

Abstract

Stochastic programming is concerned with decision making under uncertainty, seeking an optimal policy with respect to a set of possible future scenarios. This paper looks at multistage decision problems where the uncertainty is revealed over time. First, decisions are made with respect to all possible future scenarios. Secondly, after observing the random variables, a set of scenario specific decisions is taken. Our goal is to develop algorithms that can be used as a back-end solver for high-level modeling languages. In this paper we propose a scenario decomposition method to solve multistage stochastic combinatorial decision problems recursively. Our approach is applicable to general problem structures, utilizes standard solving technology and is highly parallelizable. We provide experimental results to show how it efficiently solves benchmarks with hundreds of scenarios.

Introduction

Machine learning and statistical inference methods are increasingly popular to create value from data, for example in forecasting customer demand, patient flow in healthcare, or travel times in transport systems. However, to harness the real value of predictions, one has to understand how to use this information to improve decision making. Importantly, predictions are always subject to a certain confidence level. In order to make realistic decisions, it is thus crucial to take this inherent uncertainty into account.

Modeling uncertainty when solving real-world decision problems is crucial for producing robust solutions. Oftentimes, a substantial benefit results when describing decision problems with multiple time stages (Huang and Ahmed 2009). Examples of multistage problems are the management of inventory over multiple weeks with uncertain customer demand, or production planning over a longer time horizon with uncertain processing times or due dates. Stochastic programming has been widely studied and applied as a framework to model and solve decision problems where the uncertainty is independent of the decisions made (Birge and Louveaux 2011).

Modeling frameworks, such as GAMS (McCarl, Meeraus, and Van der Eijk 2012), AIMMS (Bisschop and Roelofs 2006), AMPL (Fourer, Gay, and Kernighan 1989) or MiniZinc (Nethercote et al. 2007), have been developed to express deterministic decision problems (without presence of uncertainty). These frameworks enable a user to simply express a decision problem, irrespective of the solving approach used to determine a solution. Once modeled, a wide range of solver technologies (such as Mixed Integer Linear Programming or Constraint Programming) can be used to find solutions. As mentioned by Watson, Woodruff, and Hart (2012), the coherent chain of modeling and solving used for deterministic decision problems is not yet as well developed for their stochastic counterparts, hindering the widespread adoption of stochastic programming.

Over the last years, the mentioned modeling frameworks have been extended to support decision models that are subject to uncertainty. The uncertainty is commonly expressed using *scenarios*, where each scenario describes the stochastic problem when all the random variables are fixed. Furthermore, each scenario has a given probability of occurrence. Given a model of a stochastic problem, two solving paradigms are generally available to find solutions. First, the model can be transformed into the *extensive form*, the so called *deterministic equivalent* (DE). The DE expresses all scenarios together in one large model that can be solved by standard Mixed Integer (MIP) or Constraint Programming (CP) solvers. However, the DE does not scale well with respect to the number of scenarios or decision stages, and solving the DE is intractable for all but the smallest problem instances. Secondly, dedicated algorithms for stochastic programs have been developed. These algorithms typically exploit the model structure of stochastic programs and iteratively solve the decision problem. The publicly available solvers DECIS (Infanger 1999) and FortSP (Ellison, Mitra, and Zverovich 2010) employ the L-Shaped method (Van Slyke and Wets 1969), a variation of Benders decomposition for solving stochastic programs. Watson, Woodruff, and Hart (2012) propose to use Progressive Hedging (PH), a scenario decomposition algorithm, as a back-end solver for their modeling framework PySP. Both Progressive Hedging and the L-Shaped method are designed to solve problems with continuous variables and convex constraints; extensions exist that permit discrete variables.

The focus of this work is on scenario based multistage stochastic optimization problems with a *combinatorial structure*, i.e., discrete variables and nonlinear constraints in any stage of the problem. We propose a novel parallel search algorithm that can solve stochastic multistage problems, based on two-stage approaches recently published by Ahmed (2013). Our approach is based on standard solving technology (such as MIP or CP) and does not require any specialized model reformulation, making our algorithm a prime candidate for a back-end to high-level, expressive modeling languages. Empirically, we demonstrate the efficiency of our algorithm and show how it clearly outperforms the DE.

Background

This section introduces first an example of a multistage stochastic problem followed by the basic notation.

Example: Stochastic Facility Location

The classical facility location problem has many applications, such as deciding on where to open a warehouse in a supply chain, determining the location of a database in a computer network, or selecting the most appropriate vendors (Arabani and Farahani 2012). We motivate our work using a multi-period stochastic facility location problem with capacity constraints; Farahani et al. (2014) present a comprehensive review. The task is to open a number of facilities, assign each customer to a single facility, whilst minimizing the combined cost of setting up facilities and distributing goods. The problem has multiple stages and the number of customers (including locations) is uncertain and revealed over time. The uncertainty is characterized using scenarios, as shown in Fig. 1 for the three-stage case with four scenarios. Each scenario describes a different set of customers, which may have to be served in the future, e.g. the green (V_{21}) and red (V_{22}) customers in stage 2. In the *first stage*, before receiving information about the random variables (edges leaving V_1), a set of facilities is opened (in this example, only one facility is opened). In the *second stage*, depending on the scenario, the customers are assigned to the facilities already opened in stage one, and additional facilities are opened to cater for extra customers in stage three (node V_{21} and V_{22}). Finally, in the third stage all customers are assigned to the existing facilities, without opening new ones. The objective is to minimize the *expected* (average) cost over all scenarios.

Basic definitions

We build upon the notation we introduced earlier for the two-stage case (Hemmi, Tack, and Wallace 2017), instead of the notation used by Tarim, Manandhar, and Walsh (2006), as it more naturally captures the concept of scenario decomposition, the basis of the algorithm presented here. To simplify the presentation, we assume equal probability for all scenarios. This does not restrict the algorithm, as the probabilities are only used to compute the objective function.

Note: Scenarios with non-uniform probabilities can be unified by duplicating scenarios (e.g. if scenario A is twice as likely as scenario B, duplicating A yields three equally probable scenarios A, A', B). However, an efficient implementation uses the correct probabilities rather than duplicating the scenarios.

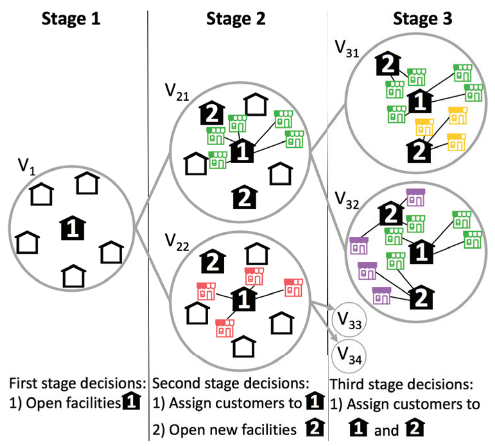


Figure 1: A three-stage stochastic facility location problem

A stochastic optimization problem can be derived from a deterministic optimization problem, which is defined as:

Definition 1 *A (deterministic)* ***constraint optimization problem*** *(COP) is a four-tuple P:*

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

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 $x_i, \ldots, x_j$, termed scope(c) and specifies mutually-compatible variable assignments σ from the Cartesian product $D(x_i) \times \cdots \times D(x_j)$. The quality of a solution is measured using the objective function f. We write scope(σ) for the variables that appear in σ; $\sigma(x)$ for the value of x in assignment σ; $\sigma|_X$ for σ restricted to the set of variables X; and $\sigma \in D$ means $\forall x : \sigma(x) \in D(x)$. We write the union of two assignments (with disjoint scopes) $\sigma_1 \wedge \sigma_2$. Furthermore, we define the set of ***solutions*** *of a COP as the set of assignments to decision variables from the domain D that satisfy all constraints in C:*

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

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

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

On the basis of a COP, we define now a two-stage Stochastic Constraint Optimization Problem (SCOP). In the two-stage case, a set of *first-stage decisions* is taken before the values of the random variables are revealed, e.g. opening one facility in V_1 in Fig. 1. Once the random variables are fixed, the second-stage decisions are taken with respect to the realization of the random variables. The objective is to find a first-stage assignment that optimizes the expected value, which is the average objective over all scenarios. We can define this formally as:

Definition 2 *A* ***two-stage stochastic constraint optimization problem*** *(SCOP) is a tuple:*

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

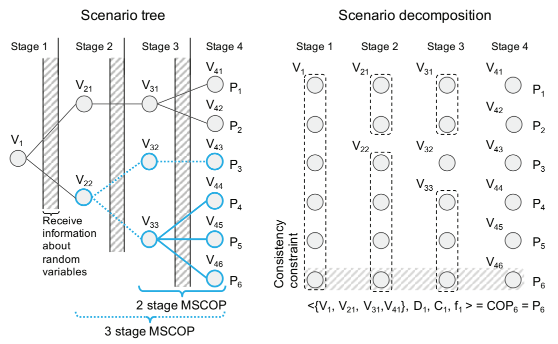


Figure 2: A four-stage stochastic optimization problem.

*with k scenarios where each P_i is a COP, and the set V is the set of first-stage variables shared by all P_i. The set of **solutions** for a two-stage SCOP is defined as the tuples of P_i solutions that agree on the shared variables V. An **optimal solution** to an SCOP minimizes the combined sum of the individual objectives (and hence the average).*

We now generalize the two-stage notation to multistage SCOPs, where in each stage decisions are made with respect to all possible future scenarios.

Definition 3 *A **multistage stochastic constraint optimization problem** (MSCOP) is a tuple:*

$$\begin{aligned}\hat{P} \quad &=< V, [\hat{P}_1, \dots, \hat{P}_n] > \\ &\textit{or} < V, P >\end{aligned}$$

where each $\hat{P}_1 \dots \hat{P}_n$ is an MSCOP and V represents the shared variables in the first (or root)-stage.

As the definition suggests, an MSCOP has a tree structure, defined by the set of decision variables in each stage. The left part of Fig. 2 displays the tree of a four-stage MSCOP. The illustration consists of six scenarios, where each scenario denotes a path from the root to a leaf node. Note that a subset of scenarios share parts of a path. For example, the paths of scenarios P_1 and P_2 only diverge after stage 3, but differ from scenario P_3–P_6 already after stage 1. This implies that the decisions taken in stage 1 are *the same* for all scenarios, yet decisions taken in stage 2 and 3 are only *consistent* for scenarios P_1 and P_2, and likewise for the scenarios P_3–P_6. We can therefore say that the four-stage problem is composed of two three-stage problems and so on. We call this representation the *scenario tree* of the problem. $V_{i,j}$ represents the shared variables of each stage. The arcs leaving the V nodes describe a concrete instantiation of the random variables. For example, the random variables of the first stage take on different sets of values, e.g. a different set of available customers in the facility location problem in Fig. 1.

A solution to a stochastic problem is a policy tree (Tarim, Manandhar, and Walsh 2006), which specifies the decisions to take in each stage depending on the preceding realization of random variables.

Definition 4 *A **policy tree** $\mathcal{T}$ is defined as*

$$\mathcal{T} \quad =< \sigma, [\mathcal{T}_1, \dots, \mathcal{T}_n] >$$

*and contains an assignment of the root-stage variables σ and a list of policy trees $[\mathcal{T}_1, \dots, \mathcal{T}_n]$, one for each branch in the scenario tree. A policy tree $\mathcal{T}$ **matches** an MSCOP $\hat{P}$, if and only if the scope of each assignment σ matches the variables of the corresponding scenario and stage:*
$\mathcal{T} =< \sigma, [\mathcal{T}_1, \dots, \mathcal{T}_n] >$ matches $\hat{P} =< V, [\hat{P}_1, \dots, \hat{P}_m] >$ iff $scope(\sigma) = V$ with $n = m$, and for all $i \in 1 \dots n$, $\mathcal{T}_i$ matches $\hat{P}_i$.

*A **path** of a policy tree $\mathcal{T}$ and a matching $\hat{P}$ is a tuple $< [\sigma_1, \dots, \sigma_d], P >$, collecting all the assignments σ_i from the root to a leaf of $\mathcal{T}$, and the COP P at the corresponding leaf of $\hat{P}$. We write $paths(\mathcal{T}, P)$ for this set of paths.*

*Finally, the set of **solutions** of an MSCOP $\hat{P}$ is defined as the matching policy trees for which each path is a solution to the underlying COP P:*

$$\begin{aligned} sol(\hat{P}) = \{ \quad & \mathcal{T} \mid \mathcal{T} \textit{ matches } \hat{P}, \\ & \forall \textit{ paths } p = < [\sigma_1, \dots, \sigma_d], P > \\ & \in paths(\mathcal{T}, \hat{P}) : \sigma_1 \wedge \dots \wedge \sigma_d \in sol(P)\} \end{aligned}$$

*An **optimal solution** to an MSCOP minimizes the sum of the individual objectives:*

$$\underset{\mathcal{T} \in sol(\hat{P})}{\arg\min} \quad \sum\nolimits_{<[\sigma_1, \dots, \sigma_d], <V,D,C,f>> \in paths(\mathcal{T}, \hat{P})} f(\sigma_1 \wedge \dots \wedge \sigma_d)$$

Multiple methods for solving stochastic optimization problems have been developed in the past. The most straightforward approach is to transform the MSCOP into the Deterministic Equivalent and use a standard MIP or CP solver to find solutions. However, as mentioned before the DE lacks scalability with regard to the number of scenarios and decision stages. Alternatively, the SCOP can be *decomposed* in one of two ways, see the work of Aldasoro et al. (2017) for a more comprehensive summary. Firstly, the problem can be relaxed by time stages and solved in a Benders decomposition fashion. A master problem describes the stage before the random variables are fixed and contains an approximation of the subsequent stages. The stage after the random variables are fixed denotes the sub-problems, one per scenario. Complete assignments to the master problem are evaluated against the sub-problems to obtain feasible solutions. This scheme is called L-shaped method (Birge and Louveaux 2011). Secondly, the problem can be decomposed by scenarios instead of time stages. This *scenario decomposition*, which is introduced in the next section, is the basis for the rest of this paper.

Recursive Evaluate and Cut

This section introduces the basic idea of scenario decomposition before explaining our main contribution, a recursive algorithm for solving stochastic multistage problems.

Scenario decomposition

In our definition of an SCOP or MSCOP, each scenario is represented as a COP, and a solution to the overall problem is a policy tree that consists of solutions to the individual scenario COPs that agree on the values of the shared variables (consistency). Scenario decomposition algorithms *relax* the requirement for all COPs to agree on the shared variables. That way, each scenario COP can be solved independently of the others. The solutions to the individual COPs potentially violate the consistency conditions for the shared variables – the optimal first-stage decision for one scenario might yield a poor objective for another scenario. Scenario decomposition algorithms such as Progressive Hedging (Rockafellar and Wets 1991), or indeed Ahmed's algorithm (described below), iteratively enforce convergence (i.e., consistency) over the shared variables to obtain a feasible solution to the MSCOP. Furthermore, the sum of the scenario objective values of the individual COPs yields a *lower bound* on the objective of the MSCOP, which can be used as a termination condition.

Two-stage algorithm

Ahmed (2013) proposes a scenario decomposition algorithm for two-stage SCOPs with binary first-stage variables and arbitrary sub-problems (i.e., not restricted to linear constraints). The consistency constraints are relaxed, and the scenario sub-problems are solved independently. The algorithm comprises three main steps: *obtain* and *evaluate* candidates solutions, then add *cuts*. In a previous paper (Hemmi, Tack, and Wallace 2017), which improves Ahmed's original two-stage algorithm, we call this method *Evaluate and Cut* (E&C). In the following, we present the two-stage algorithm again and then extend it to support multiple stages.

Algorithm 1 explains in pseudo code how E&C works for two-stage problems. First, the individual scenario COPs are retrieved from the scenario tree (line 3). Each COP is solved individually using a standard MIP or CP solver (line 8). The solution σ denotes an assignment to the first- and second-stage variables and the sum of the scenario objectives (obj) yields a lower bound on the SCOP. The first-stage assignments σ_V of each scenario COP, called *candidates*, are *evaluated* against all other scenarios by projecting their variable assignment onto the first-stage variables of the other scenarios (line 16). Adding up the objectives that result from the candidate evaluation yields an upper bound. Finally, the evaluated first-stage assignment σ_V is cut (pruned) from the search by adding a *nogood* constraint to each of the scenario COPs (line 18). By iteratively repeating the three steps – obtaining, evaluating and cutting candidates – the procedure is guaranteed to find an optimal solution, as long as the first-stage variables have finite domains. Completeness holds because the lower bound is monotonically increasing as a result of cutting off the evaluated solutions. Optimality is proven once the lower bound meets the upper bound. The scenarios can be evaluated independently, allowing highly parallelized implementations. Experiments conducted by Ahmed (2013), Ryan, Rajan, and Ahmed (2016) and ourselves (Hemmi, Tack, and Wallace 2017) confirm the effectiveness of E&C.

Algorithm 1 Evaluate and Cut for Two-Stage Problems

```
1:  procedure SOLVESCOP($\hat{P}$)
2:      Initialize: UB = $\infty$, LB = -$\infty$, sol = null
3:      [P$_1$,...,P$_k$] = GET_SCENARIOS($\hat{P}$)
4:      while LB $<$ UB do
5:          LB = 0, S= $\varnothing$
6:          % Find first-stage candidates
7:          for i in 1..k do
8:              $<\sigma$,obj$>$ = SOLVE(P$_i$)
9:              LB += obj
10:             S $\cup = \{\sigma|_V\}$
11:         % Evaluate first-stage candidates
12:         % to obtain an upper bound
13:         for $\sigma_V \in$ S do
14:             t$_{UB}$ = 0
15:             for i in 1..k do
16:                 $<$ _,obj$>$ = SOLVE(P$_i$[C $\cup = \{\sigma_V\}$])
17:                 t$_{UB}$ += obj
18:                 ADDNOGOOD(P$_k$,$\sigma_V$)
19:             if t$_{ub} <$ UB then
20:                 sol = $\sigma_V$
21:                 UB = t$_{UB}$
22:     return sol
```

Recursive multistage E&C

We will now extend the two-stage E&C algorithm into a recursive multistage version, the main contribution of this paper. Just like for two-stage problems, the MSCOP is decomposed into its individual scenario COPs. In contrast to the two-stage case where each COP was composed of first- and second-stage variables, the multistage COPs are composed of d variables V$_d$, one for each stage. Since the scenarios now form a tree, each scenario COP contains variables that need to be consistent with some other scenarios. This is illustrated in Fig. 2 on the right side, where for example the COP for scenario 6 has variables V_1, V_{22}, V_{33} and V_{46}; V_1 needs to be consistent with all other scenarios; V_{22} with scenarios 3–6; V_{33} with scenarios 4–6; and V_{46} only exists in scenario 6.

The main steps of the multistage algorithm are conceptually coherent with the two-stage algorithm. First, each scenario COP is solved individually. The solution is an assignment to all variables along the scenario path $V_1 - V_d$. Secondly, the sum of all the scenario objectives yields a lower bound. Thirdly, candidates are evaluated against all other scenarios by invoking the algorithm recursively. Finally, as in the two-stage case, each evaluated candidate is cut off from the search by adding a nogood.

Algorithm 2 implements the recursive E&C procedure and Fig. 3 supports the explanation. First, the MSCOP is unpacked into scenario COPs (line 8). The decomposed COPs are solved individually (line 13). This provides a set of root-stage candidates. By root stage we refer to the uppermost stage with non-fixed variables V, as specified in Fig. 3. The sum of the COP objectives is a lower bound for the root

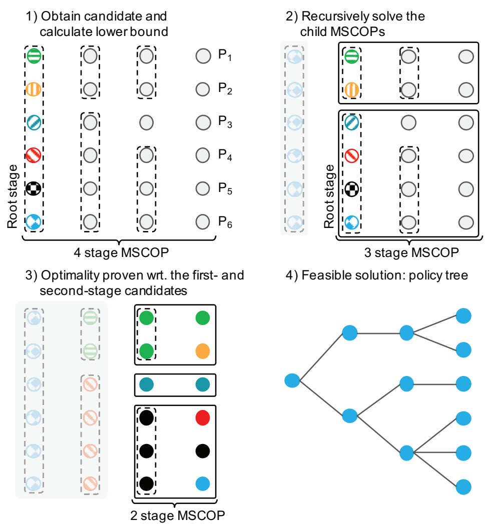

1) Obtain candidate and calculate lower bound
2) Recursively solve the child MSCOPs
P1
P2
P3
P4
P5
P6
Root stage
4 stage MSCOP
Root stage
3 stage MSCOP
3) Optimality proven wrt. the first- and second-stage candidates
4) Feasible solution: policy tree
2 stage MSCOP

Figure 3: Recursive multistage *Evaluate and Cut*.

stage. Then the candidates are evaluated against all child MSCOPs, e.g. the initial 4-stage MSCOP is composed of two 3-stage children (Fig. 3 top row). The child MSCOPs are evaluated by recursively calling the solving procure with each child $\hat{P}_i$ (line 26). The root-stage candidates are passed on to the children (as $\sigma_V \wedge \sigma_P$), to fix the variables V of the previous stages.

The recursive call returns the optimal solution $\mathcal{T}_i$ for each child with respect to the current candidate (Fig. 3 bottom left solid fill). After the recursion is finished, an upper bound for the first stage with a resulting policy tree $\mathcal{T}$ is constructed. The nogoods generated during the recursive call are valid globally and stay active after the recursion has finished (line 32).

Implementation and Improvements

Memorization: At every recursion level, a candidate is evaluated against all the scenarios that belong to the MSCOP (Algorithm 2, line 13). At least one, and possibly many scenario COPs already explored a solution that matches that candidate, which results in redundant computations. This redundancy can be avoided by memorizing and mapping solutions appropriately.

Up-front candidate elimination: Before starting a recursion, it is possible to compute a lower bound for each of the candidates (Algorithm 2, line 19). If the lower bound of a candidate exceeds the incumbent objective, a nogood excluding the assignment is directly added to all relevant scenarios without entering the recursion. Otherwise, the algorithm proceeds as described earlier. Note that in the two-stage case this optimization is in fact equivalent to the candidate evaluation, but for multistage problems it can lead to significant time savings. Algorithm 2 is slightly simplified, the actual implementation makes sure that the same candidate COP is not solved multiple times using memorization as mentioned before.

Algorithm 2 Recursive Multistage Evaluate and Cut

```
 1: procedure SOLVEMSCOP($\hat{P}$, $\sigma_p$)
 2:     if <V,P> = $\hat{P}$ then
 3:         < $\sigma$, obj > = SOLVE(P[C $\cup = \{\sigma_p\}$])
 4:         $\mathcal{T}$ = < $\sigma|_V$, [] >
 5:         return < $\mathcal{T}$, obj >
 6:     else <V,[$\hat{P}_1,\ldots,\hat{P}_n$]> = $\hat{P}$
 7:         Initialize: UB = $\infty$, LB = 0, S= $\varnothing$
 8:         [P$_1$,...,P$_k$] = GET_SCENARIOS($\hat{P}$)
 9:         while LB < UB do
10:             % Obtain a lower bound
11:             % and find candidate solutions
12:             for i in 1..k do
13:                 < $\sigma$,obj> = SOLVE(P$_i$[C $\cup = \{\sigma_p\}$])
14:                 LB += obj
15:                 S $\cup = \{\sigma|_V\}$
16:             % IMPROVEMENT:
17:             % Obtain candidate lower bound
18:             % for up-front candidate elimination
19:             CLB = GETCNDLB(S,[P$_1$,...,P$_k$],$\sigma_p$)
20:             % Evaluate candidates
21:             for $\sigma_V \in$ S do
22:                 if CLB[$\sigma_V$] < UB then
23:                     t$_{UB}$ = 0
24:                     for i in 1..n do
25:                         < $\mathcal{T}_i$, obj > =
26:                         SOLVEMSCOP($\hat{P}_i$, $\sigma_V \wedge \sigma_p$)
27:                         t$_{UB}$ += obj
28:                     if t$_{ub}$ < UB then
29:                         UB = t$_{UB}$
30:                     sol = < $\sigma_V$, [$\mathcal{T}_1$, .., $\mathcal{T}_n$] >
31:                 for i in 1..k do
32:                     ADDNOGOOD(P$_i$,$\sigma_V \wedge \sigma_p$)
33:         return sol< sol, UB >
34:
35: procedure GETCNDLB(S,[P$_1$,...,P$_k$],$\sigma_p$)
36:     Initialize: CLB = new map from assignments to int
37:     for $\sigma_V \in$ S do
38:         CLB[$\sigma_V$] = 0
39:         for i in 1..k do
40:             < _, obj > = SOLVE(P$_i$[C $\cup = \{\sigma_V \wedge \sigma_p\}$])
41:             CLB[$\sigma_V$] += obj
42:     return CLB
```

Experiments

To evaluate the performance of the proposed algorithm we are using two variations of the stochastic facility location problem introduced earlier. Equation 1 contains the model for a single scenario. The goal is to minimize the combined cost of setting up facilities and delivering goods to customers, where f_{it} denotes the cost of setting up facility y_i in stage t and c_{tij} is the cost of delivering goods from facility i to customer j. Each customer is assigned to exactly one facility that might be different in each stage. C_i is the total capacity of facility i and d_j the demand of customer j. The second variant of the problem contains an additional "balancing" constraint that enforces a similar customer count for each facility. The stochastic problem takes on a form similar to the scenario tree displayed in Fig. 2, yet with all MSCOPs being fully balanced (the number of scenarios for each sub-MSCOP is equal). Overall, the aim is to minimize the expected cost while satisfying the consistency constraints.

For the experiments we generated 15 four-stage instances with 7 scenarios in stages 1 to 3, to end up with a total of 343 scenarios. For the 15 instances, we extracted further 9 problems, each containing a subset of the 343 scenarios, chosen such that a balanced tree would result, e.g. 3 scenarios in each stage would yield 27 scenarios. A total count of a 150 problems ranging from 27 to 343 scenarios was used for each of the problem classes. The instance parameters are: 6 facilities, 150 customers, the total warehouse capacity is 60% higher that the maximal customer demand, K_3 is set to 15 and K_1 and K_2 are set to 150. The balancing constraint is only enforced in stage 4, as preliminary studies revealed that the balancing constraint poses significant difficulties for the solver (CPLEX) to solve the DE, even if enforced only in one stage.

The instances were modeled using MiniZinc, both for the DE and for the individual scenario COPs in E&C. The DE was solved using CPLEX (12.6.3) parallel optimizer and the recursive E&C was implemented in Python 2.7 with CPLEX (12.6.3) as the sub-problem solver. The experiments where carried out on a single computer that is part of the MonARCH HPC Cluster provided by Monash eResearch Centre with 16 physical cores (32 hyper threaded cores) at 3.20 GHz, with a time out of 1800 seconds per instance.

Basic model

$$\min\Big\{ \sum_{t\in T, i\in I} f_{ti} * y_{ti} + \sum_{t\in T} \sum_{i\in I, j\in J} c_{tij} * x_{tij} \Big\}$$

$$s.t. \sum_{i\in I} x_{tij} = 1 \qquad \forall j \in J, t \in T$$

$$\sum_{j\in J} x_{tij} * d_j \leq C_i \qquad \forall i \in I, t \in T$$

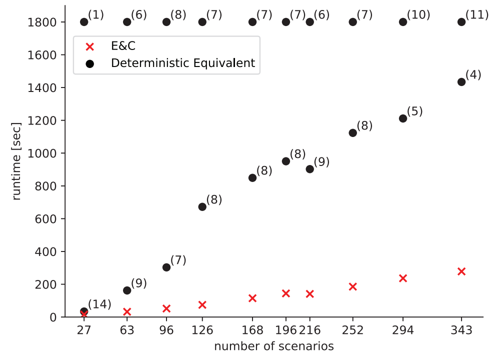
$$x_{tij} \geq 0, y_{ti} = \{0,1\} \qquad \forall t \in T, i \in I, j \in J$$

Balancing constraint

$$\Big| \sum_{j\in J} x_{ti_1 j} - \sum_{j\in J} x_{ti_2 j} \Big| < K_t \qquad \forall i_1, i_2 \in I, t \in T \tag{1}$$

Results

Basic Model: Fig. 4 shows the time to solve the basic facility location problem using E&C and CPLEX on the DE. As expected, CPLEX is able to solve the DE for small instances in a reasonable time but does not scale well with the number of scenarios, as witnessed by the large number of instances that did not finish the search within the time out. For example, when looking at the instances with 343 scenarios, 11 out of 15 problems did not finish within the given 1800 seconds. In contrast, the E&C algorithm always finishes within the time out. The run time of E&C increases close to linearly with the number of scenarios per instance.



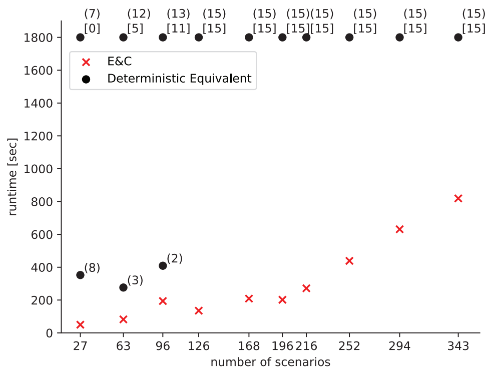
Average run time of 15 instances for each number of scenarios.
(*) number of instanced that either reach the time out of 1800 seconds (top) or find and proof the optimal solution (bottom).
E&C never times out.

Figure 4: Time vs. number of scenarios (basic model)

Model with extra constraint: Fig. 5 shows the time to solve the facility location problem with balancing constraints using E&C and CPLEX on the DE. The additional constraints make it substantially harder for CPLEX to solve the DE, with only few instances solved to optimality within the time out. For most DE instances CPLEX cannot even determine an initial feasible solution (number in square brackets in Fig. 5). Proving optimality is a great challenge for CPLEX. This can even be observed when solving individual scenarios. For E&C, we do not require a full proof of optimality for each scenario COP – all we need is a lower bound. Therefore, setting the relative optimality gap in CPLEX to 10^{-3} (instead of the default value of 10^{-4}) makes each scenario COP terminate much faster, without compromising on the completeness of the overall algorithm. The results in Fig. 5 clearly demonstrate the effectiveness of E&C. While CPLEX cannot cope with the DE formulation except in 13 out of 150 instances, recursive multistage E&C can find the optimal solution for all 150 instances within the time out.

Related Work

Challenges in solving multistage stochastic optimization problems with integer variables are well documented (Schultz 2003). Various research directions have been explored, however a substantial amount of work has focused on decomposition algorithms. One line of work has developed algorithms to decompose the problems by stages. An example of this decomposition is the L-shaped method (Van Slyke and Wets 1969) inspired by the well known Benders decomposition. Solver systems such as DECIS (Infanger 1999) and FortSP (Ellison, Mitra, and Zverovich 2010) implement the nested L-Shaped method to solve multistage problems. However, those solvers are restricted to solve problems with linear constraints and continuous variables. In contrast, recursive E&C can be used to solve problems with combina-



(*) number of instanced that either reach the time out of 1800 seconds (top) or find and proof the optimal solution (bottom).
[*] number of instances that did not find an initial solution.
E&C never reaches time out. For each scenario group, 15 instances where solved and displayed is the average time.

Figure 5: Time vs. number of scenarios (extended model)

torial structure.

Another line of work is based on the scenario decomposition. Watson, Woodruff, and Hart (2012) propose to use Progressive Hedging (PH) as a back-end solver for their modeling framework PySP. Rockafellar and Wets (1991) originally proposed the PH methodology to solve convex problems. The drawback using PH is twofold; first, optimality is only guaranteed for continuous convex problems; secondly, PH is not as flexible as recursive E&C, as it requires parameter tuning that can be expensive in practice. Alonso-Ayuso, Escudero, and Ortuno (2003) introduce an algorithm that relaxes the integrality requirements in addition to the consistency constraints, named Branch-and-Fix (BF). The scenarios are solved using a linear programming based branch-and-bound (b&b) procedure. To enforce consistency, a common branching tree is used to fix nodes in the scenario b&b tree. Aldasoro et al. (2017) have introduced a parallel BF coordination scheme that increases performance but does not remain complete. To the best of our knowledge, BF is not publicly available, its implementation is not trivial and it is not clear how to implement it in a generic fashion to solve combinatorial problems.

Scenario clustering, a line of research orthogonal to scenario decomposition, aims to tighten the bounds by bundling multiple scenarios into a DE. Clustering reduces the number of sub-problems by strategically enforcing a subset of the consistency constraints throughout the entire search procedure. It capitalizes on the fact that the DE can be solved reasonably fast for a small number of scenarios. The effectiveness of scenario clustering has been demonstrated in Sandikci and Özaltın (2014), Aldasoro et al. (2017), Escudero, Garín, and Unzueta (2016), amongst others. To incorporate scenario clustering into the E&C methodology is straightforward, however has not been done for this paper as the focus shall be on the newly introduced algorithm.

Ahmed (2013) originally introduced E&C to solve two-stage problems. Ryan, Rajan, and Ahmed (2015) proposed an asynchronous, distributed implementation of the algorithm and various improvements, such as optimality cuts (using duality) and a lower bound on the candidate evaluation based on the linear programming relaxation of the problem. These improvements, while proposed for two-stage problems, can be incorporated into our recursive algorithm. In a previous paper (Hemmi, Tack, and Wallace 2017) we extended two-stage E&C by "diving", a strategic approach to generate *partial nogoods* that perform stronger pruning than the candidate nogoods used in standard E&C. The idea is to partially enforce consistency over the first-stage variables when generating candidates. If the resulting bound is not better than the incumbent, it is guaranteed that none of the resulting candidates will improve the incumbent objective and therefore all candidates that extend on the enforced partial consistency can be pruned from the search. The up-front candidate elimination used in recursive E&C, where a lower bound on each candidate is computed prior to starting a new recursion also yields partial candidate nogoods, as a candidate might be eliminated from the search before reaching the bottom of the recursion. However, in contrast to diving we have not reasoned over partial assignments to a time-stage, an extension that would be interesting to consider.

Conclusion and Future Work

The main contribution of this paper is the introduction of a search algorithm to solve stochastic combinatorial multistage problems. Unlike other algorithms, our method can be applied to solve problems with complex structure such as non-linear constraints over integer variables. No complicated problem reformulation is required making our method a prime candidate to be used as back-end solver for modeling frameworks such as GAMS, AMPL, AIMMS or Stochastic MiniZinc (Rendl, Tack, and Stuckey 2014). We have demonstrated the effectiveness of recursive E&C on two sets of benchmarks, by comparing E&C with the DE formulation, the only available and applicable solver option. For future work, it will be interesting to investigate how information learned in one scenario, e.g. feasibility cuts, can be generalized and used in other scenarios, to improve the lower bound computations. Furthermore, ideas from logic-based benders decomposition (Hooker and Ottosson 2003) and branch-and-check (Thorsteinsson 2001) might lend themselves well to combine scenario decomposition with stage wise decomposition.

References

Ahmed, S. 2013. A scenario decomposition algorithm for 0–1 stochastic programs. *Operations Research Letters* 41(6):565–569.

Aldasoro, U.; Escudero, L. F.; Merino, M.; and Pérez, G. 2017. A parallel branch-and-fix coordination based matheuristic algorithm for solving large sized multistage stochastic mixed 0–1 problems. *European Journal of Operational Research* 258(2):590–606.

Alonso-Ayuso, A.; Escudero, L. F.; and Ortuno, M. T. 2003. Bfc, a branch-and-fix coordination algorithmic framework for solving some types of stochastic pure and mixed 0–1 programs. *European Journal of Operational Research* 151(3):503–519.

Arabani, A. B., and Farahani, R. Z. 2012. Facility location dynamics: An overview of classifications and applications. *Computers & Industrial Engineering* 62(1):408–420.

Birge, J. R., and Louveaux, F. 2011. *Introduction to stochastic programming*. Springer Science & Business Media.

Bisschop, J., and Roelofs, M. 2006. *AIMMS - Language Reference*. Lulu.com.

Ellison, F.; Mitra, G.; and Zverovich, V. 2010. Fortsp: a stochastic programming solver. *OptiRisk Systems*.

Escudero, L. F.; Garín, M. A.; and Unzueta, A. 2016. Cluster lagrangean decomposition in multistage stochastic optimization. *Computers & Operations Research* 67:48–62.

Farahani, R. Z.; Hekmatfar, M.; Fahimnia, B.; and Kazemzadeh, N. 2014. Hierarchical facility location problem: Models, classifications, techniques, and applications. *Computers & Industrial Engineering* 68:104–117.

Fourer, R.; Gay, D. M.; and Kernighan, B. 1989. Algorithms and model formulations in mathematical programming. New York, NY, USA: Springer-Verlag New York, Inc. chapter AMPL: A Mathematical Programming Language, 150–151.

Hemmi, D.; Tack, G.; and Wallace, M. 2017. Scenario-based learning for stochastic combinatorial optimisation. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer.

Hooker, J. N., and Ottosson, G. 2003. Logic-based benders decomposition. *Mathematical Programming* 96(1):33–60.

Huang, K., and Ahmed, S. 2009. The value of multistage stochastic programming in capacity planning under uncertainty. *Operations Research* 57(4):893–904.

Infanger, G. 1999. Gams/decis user's guide.

McCarl, B.; Meeraus, A.; and Van der Eijk, P. 2012. Mccarl expanded gams user guide version 23.6. *GAMS Development, Washington, DC*.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming*. Springer. 529–543.

Rendl, A.; Tack, G.; and Stuckey, P. J. 2014. Stochastic minizinc. In *Principles and Practice of Constraint Programming*, 636–645. Springer.

Rockafellar, R. T., and Wets, R. J.-B. 1991. Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of operations research* 16(1):119–147.

Ryan, K.; Rajan, D.; and Ahmed, S. 2015. Scenario decomposition for 0-1 stochastic programs: Improvements and asynchronous implementation. *Available at Optimization-Online http://www. optimization-online. org/DB_FILE/2015/11/5201. pdf*.

Ryan, K.; Rajan, D.; and Ahmed, S. 2016. Scenario decomposition for 0-1 stochastic programs: Improvements and asynchronous implementation. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, 722–729. IEEE.

Sandikci, B., and Özaltın, O. Y. 2014. A scalable bounding method for multi-stage stochastic integer programs.

Schultz, R. 2003. Stochastic programming with integer variables. *Mathematical Programming* 97(1):285–309.

Tarim, S. A.; Manandhar, S.; and Walsh, T. 2006. Stochastic constraint programming: A scenario-based approach. *Constraints* 11(1):53–80.

Thorsteinsson, E. 2001. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In *Principles and Practice of Constraint Programming—CP 2001*, 16–30. Springer.

Van Slyke, R. M., and Wets, R. 1969. L-shaped linear programs with applications to optimal control and stochastic programming. *SIAM Journal on Applied Mathematics* 17(4):638–663.

Watson, J.-P.; Woodruff, D. L.; and Hart, W. E. 2012. Pysp: modeling and solving stochastic programs in python. *Mathematical Programming Computation* 4(2):109–149.