Computational Management Science (2020) 17:277–307
https://doi.org/10.1007/s10287-020-00369-2

ORIGINAL PAPER

Scenario tree construction driven by heuristic solutions of the optimization problem

Vit Prochazka[1,2] · **Stein W. Wallace**[1]

Received: 4 November 2019 / Accepted: 10 April 2020 / Published online: 20 June 2020

Abstract
We present a new scenario generation process approach driven purely by the out-of-sample performance of a pool of solutions, obtained by some heuristic procedure. We formulate a loss function that measures the discrepancy between out-of-sample and in-sample (in-tree) performance of the solutions. By minimizing such a (usually non-linear, non-convex) loss function for a given number of scenarios, we receive an approximation of the underlying probability distribution with respect to the optimization problem. This approach is especially convenient in cases where the optimization problem is solvable only for a very limited number of scenarios, but an out-of-sample evaluation of the solution is reasonably fast. Another possible usage is the case of binary distributions, where classical scenario generation methods based on fitting the scenario tree and the underlying distribution do not work.

Keywords Stochastic optimization · Scenario tree · Scenario generation

1 Introduction

Most methods for solving stochastic programs require discrete scenarios as input. Exceptions would be simple (often inventory) models that have closed-form solutions and methods such as stochastic decomposition (Higle and Sen 1991), where the discretization takes place inside the method. The simplest way to find scenarios that can be used as input is normally to sample (Cario and Nelson 1997; Lurie and Goldberg 1998). If sampling leads to numerically solvable problems with high enough

✉ Vit Prochazka
vit.prochazka@snf.no

Stein W. Wallace
stein.wallace@nhh.no

1 Department of Business and Management Science, NHH Norwegian School of Economics, Helleveien 30, Bergen, Norway

2 SNF – Centre for Applied Research at NHH, Helleveien 30, Bergen, Norway

accuracy in short enough time [see for example the discussion in Kaut et al. (2007)], there is no good reason to do anything more complicated. But, if for some reason, sampling is not acceptable, there is a need for something more advanced—and most likely more computationally challenging. Very often, the alternatives require some rather expensive offline computations, but lead to a much more efficient online performance. For an overview, see for example King et al. (2012). The methods fall into two major classes; those that try to approximate the probability distribution itself, and those that focus on the quality of the solutions that emerge from using the scenarios. We can call these methods *distribution-oriented* and *problem-oriented*. Both use a metric to measure distance, the first uses measures from probability theory (such as the Kantorovich–Rubinstein or Wasserstein metric Pflug 2001), the second uses the optimization problem itself as metric. This paper falls into the second category, and is hence connected to the principal thinking in Høyland and Wallace (2001), and the methodology set out by Fairbrother et al. (2017). But contrary to the latter work, we do not need to analyze the optimization problem itself, rather we need to be able to produce a set of feasible solutions and to perform, rather efficiently, out-of-sample objective function evaluations.

Hence, in this paper, we introduce a framework that enables generating scenarios in a problem-oriented fashion, but without analyzing the problem explicitly. Our general approach is based on a pool of solutions and a measure of discrepancy between in-sample (in-tree) and out-of-sample performance of the solutions, which we aim to minimize. By *solution* we mean some vector of decision variables that can be used for the evaluation of the problem's objective function. Thus, the solution does not have to be optimal (and most likely it is not), nor feasible in scenario-related constraints (discussed in Sect. 2.5). A solution is generated by some heuristic with a reasonable trade-off between speed and accuracy. Every solution from the pool can be evaluated out-of-sample, that is, we can determine its "true" value by using the underlying distribution, and in-sample (in-tree) by using the corresponding scenario tree. We define a loss function that measures the discrepancy between out-of-sample and in-sample performance of the pool of solutions. We search for a tree that minimizes the loss function.

Since we offer a general framework that requires several problem-specific subroutines, a direct comparison with other methods for generating scenarios is not easy. Such comparisons can always be distorted in (or out of) favor, by applying, for example, a different heuristic. Our framework also may require a significant time for development. Both the subroutines—the heuristic for obtaining solutions and the loss function minimization procedure—must be tailored-made for a specific problem. That is the main disadvantage compared to some other methods for scenario generation, for instance copula-based heuristics (Kaut 2014), which can be applied immediately by using a publicly released code, and which requires either historical data or specifications of the probability distribution.

Thus, rather than directly competing with these methods on solvable problems, we try to identify their limits, for example how they handle binary random variables, such as a random appearance of customers in vehicle routing problems. We offer an approach that can overcome some of the limits, and which can be applied on a larger spectrum of applications. We also hope to offer an original perspective on the rela-

tionship between uncertainty and its representation by scenarios within optimization problems.

2 Framework

Since this paper is primarily conceptual, and not technical, we will not introduce exact mathematical definitions of all elements of our procedure in order to keep the work easy to follow. In general, we assume there is a true[1] random vector ξ that enters a process to be optimized. The distribution is either parametrically described or empirically given, for example by historical data. For simplicity, we focus on two-stage problems to avoid the complications that multi-stage problem formulations and conditional distributions between stages bring. See Birge and Louveaux (1997), Kall and Wallace (1994) for proper definitions, if needed.

Without loss of generality we assume a maximization problem throughout the text

$$\max_{x \in \mathcal{X}} f(x, \xi) \tag{1}$$

where $\mathcal{X}$ is the feasible region for decisions x and f encapsulates some reward functions and their ranking criteria (expected value, value-at-risk, etc.).

A difficulty that often arises is that (1) cannot be solved when using ξ directly due to its size (in the case of an empirical distribution) or its computational intractability (in the case of a parametrically defined distribution). Thus, we search for a representation of the original distribution by a so-called scenario tree $\mathcal{T}$ consisting of particular scenarios, which are vectors of realizations of random variables for example volumes of the items in the stochastic knapsack problem), and weights[2] p associated with each scenario. Then, we solve an optimization problem

$$\max_{x \in \mathcal{X}} f(x, \mathcal{T}) \tag{2}$$

and hope that the solution of this program is also a good solution to the original problem (1). The quality of the solution and its relation to the original program can be tested, see Kaut and Wallace (2007).

Although it is almost always impossible to solve problem (1), it is often possible to evaluate the quality of a fixed solution $\hat{x}$ by using the whole distribution. That means to determine the value $f(\hat{x}, \xi)$. If it is not possible to do this exactly, then, most of the time, it can be done approximately, with very high accuracy, by using a very large sample from the true distribution. We call this procedure *out-of-sample* evaluation of the solution. Similarly, we can get an *in-sample* value $f(\hat{x}, \mathcal{T})$ for the solution $\hat{x}$ by inserting the scenario tree into the model.

Our framework to construct a scenario tree consists of two steps:

[1] In a large majority of applications, this still means highly subjective descriptions of the random phenomena.

[2] In other methods, p denote probabilities of particular scenarios. However in our approach, we do not demand their sum to be 1. Thus, we avoid calling them probabilities.

1. Heuristically generate a pool of solutions for the optimization problem; evaluate the solutions out-of-sample.
2. Construct a scenario tree in such a way that the discrepancy between in-sample and out-of-sample performance of the solutions is minimized.

The first step of our approach is to generate a pool of solutions $\mathcal{A}$ by a problem-specific heuristic and evaluate the solutions out-of-sample. This is not always possible, since for some problems even finding a feasible solution or its out-of-sample evaluation can be too difficult.[3] But our approach applies to a large class of problems, for which reasonable heuristics exist. To make the right choice of heuristic, one needs to take into consideration a trade-off between the number of solutions in the pool, accuracy of the heuristic and the time spent in this phase.

The heuristically obtained solutions are evaluated out-of-sample, either as a part of the heuristic procedure itself or afterwards, for example in the case the heuristic is using just a small sample of the original distribution.

Even though the out-of-sample evaluation might be computationally intensive procedure in some cases, as it might require solving a difficult optimization task for each of the out-of-sample scenarios, in many applications the out-of-sample evaluation is computationally much less demanding than solving the original problem itself. Very often the optimization problem to solve in the second stage is computationally easier than the problem from the first stage. For example, in network design, the first stage problem is an integer (nonlinear, non-convex) program, whereas in the second stage a decision maker faces a network flow problem (linear program). Moreover, all the programs that need to be solved in the second stage might be very similar to each other as they differ only in the vector of scenarios, which can be further exploited to speed up the computation as it is shown in Haugland and Wallace (1988).

One of the fields where we see a potential applicability of our framework are problems with binary distributions (we dedicate Sect. 2.4 to this class of problems). Then it generally takes 2^n scenarios, where n is the number of random variables, to get the exact out-of-sample value. In Prochazka and Wallace (2018), it is shown that many problems from this field have a special structure, which can be exploited to substantially (in some cases) reduce the number of scenarios needed to consider, hence speed up the out-of-sample evaluation.

To summarize this, our approach aims to problems that are difficult in some aspects (discussed in Sect. 4) but where the out-of-sample evaluation is not the crucial obstacle. We assume that the out-of-sample evaluation can be performed within a reasonable time for the entire pool of solutions.

To measure discrepancy between in-sample and out-of-sample performance of the solutions from the pool, we define a *loss function*. The function is derived from our requirements on a good scenario tree, which are discussed in the following section. For a given pool of solutions, the loss function is a function of the scenario tree. Thus we search for a scenario tree that minimizes the loss function. In Sect. 2.3, we discuss different settings of the minimization procedure, but in the case both scenarios and their weights p are considered as free variables, we deal with non-linear and non-convex

[3] For example the problem of minimum Hamiltonian cycle, where even finding a Hamiltonian cycle in a given graph is NP-complete (Garey and Johnson 1979).

functions to be minimized. This is not a trivial task, and even though there are some solvers for non-linear optimization available, they often require some problem-specific adjustments.

Our framework is essentially a heuristic, because it assumes that a good fit between in-sample and out-of-sample performance of the pool of solutions implies a good fit on the entire search space $\mathcal{X}$. This does not have to be valid generally, especially if the size of the pool is small or the heuristic for generating solutions is "biased" (only solutions with certain properties[4]) in some sense. Thus, the validity of the assumption needs to be further tested.

2.1 Properties of a good scenario tree

In this section, we discuss our views on *what makes a scenario tree good.* We list a set of requirements on the scenario tree and its relation to the original optimization problem (1). These requirements are used to formulate the loss function. Let us first focus on relatively complete recourse; handling potential infeasibilities in the second-stage problem is discussed in Sect. 2.5.

In theory, to call a scenario tree $\mathcal{T}$ (almost) perfect, we would simply need that problem (1)—had it been solvable—and (2) return the same value for the objective function of the optimal solution. There are, however, two problems. First, there are cases for which even the optimization program (2) is computationally intractable and we need to settle with some non-optimal solution, which can be reasonably good for $\mathcal{T}$, but arbitrarily bad for ξ. Hence, we put "(almost)" in the first sentence of the paragraph. Second, and more importantly, to assure that this requirement will hold is not achievable. It would require the knowledge of the optimal solution of the program for ξ. If we were able to solve the problem for ξ and get the optimal solution, generation of the scenario tree is, obviously, of more marginal interest (though it depends on the requirements on CPU time).

Let us, then, discuss what we expect from a good tree, not a perfect one. Here we summarize our requirements on a good scenario tree $\mathcal{T}$ in relation to the true distribution ξ when we perform out-of-sample and in-sample evaluation of a pool of solutions.

1. The *ranking is approximately preserved*.[5] That is, if one solution x_1 is better than x_2 when evaluated out-of-sample, it is going to be "very likely" better when evaluated in-sample.
2. We *do not observe overconfident outliers*. We argued in Requirement 1 that it is impossible to have a guarantee of the perfect ranking, so we expect it may happen that $f(x_1, \xi) > f(x_2, \xi)$, but $f(x_1, \mathcal{T}) < f(x_2, \mathcal{T})$ for some x_1 and

[4] For example, a "nearest neighbor" heuristic for a traveling salesman problem with stochastic travel times may generate only routes with relatively short distances between customers. Thus, travel times on intermediate and long distance edges might be assigned randomly (since none of those edges is included in the pool), and thus, arbitrarily distort the final results.

[5] We would naturally prefer the perfect ranking. Then, for any subset of solutions, solving $\max_x f(x, \mathcal{T})$ and $\max_x f(x, \xi)$ would be equivalent tasks with respect to our objective. But having a requirement on reaching the perfect ranking of solutions is meaningless, since it implies the ability to solve the problem $\max_x f(x, \xi)$. Usage of the scenario tree is, then, redundant.

x_2. Such a case is acceptable when the values $f(x_1, \xi)$ and $f(x_2, \xi)$ are close to each other. But we want to avoid the case where a particularly bad solution (out-of-sample) performs well in-sample, that is, its in-sample value is a (massive) overestimation of the true value. The opposite case—a truly good (out-of-sample) solution performs really badly in-sample—is not so critical, if it does not hold for many solutions. We call these outliers acceptable.

3. There is a greater emphasis on Requirements 1 and 2 to be satisfied for better solutions than for worse solutions. In other words, the scenario tree approximates better the underlying distribution in the space of higher-quality solutions, where an optimization algorithm, either an exact or a heuristics one, will search for the best solution to the program (2).
4. *In-sample values approximate well out-of-sample values*, that is $f(x_a, \xi) \approx f(x_a, \mathcal{T})$. In theory, this requirement is not needed at all. A scenario tree that can produce in-sample values of solutions that are totally off, but ranks the solutions approximately correctly, is still a very useful tree, because it enables us to find a very good solution. Then, the real value of the solution can be found by an out-of-sample evaluation.
   However, we still have this requirement on our list, not just because it is a nice (but not necessary) property of the scenario tree, but because it implies, to some extent, other requirements. If the in-sample values approximated out-of-sample values perfectly, it would also preserve the perfect ranking. Thus, we use this requirement as a starting point for the loss function formulation in 2.2.

Demonstration of the requirements of a good scenario tree

We illustrate the relationship between in-sample and out-of-sample values over the artificially created pool of solutions in Fig. 1. We show four examples of different scenario trees, on which we comment some of our views formulated above. We assume the pool consists of six solutions, sorted in ascending order according to their out-of-sample values (y-axis). We assume a maximization problem, thus the higher the out-of-sample value, the better the solution is.

Let us assume we have a scenario tree that returns out-of-sample and in-sample values as in 1a. If these six solutions were the only feasible solutions of the optimization problem, solving the program (2) would return the true optimal solution, which is fine. However, we see the tree is not reliable for other solutions and their ranking, for example solution no 1 (worst out-of-sample) is ranked as the second best one in-sample. Sometimes, problem (2) can be solved only heuristically, so even if we had guaranteed the basic property that the optimal solution in-tree is optimal out-of-sample, the heuristic could miss it and return an arbitrarily bad solution (out-of-sample).

In 1b, we demonstrate the concept of overconfident (solution no 3) and acceptable (solution no 6) outliers. If we solve $\max_x f(x, \mathcal{T})$ over this set of solutions, we determine solution no 3 as the optimal one, but it performs quite poorly in reality (out-of-sample). Had it not been for solution no 3, we would end up with solution no 5 as the optimal one. That means we would miss the true optimal solution no 6 and some related value, but the error is not as significant as for overconfident outliers, whose in-sample value is overestimated.

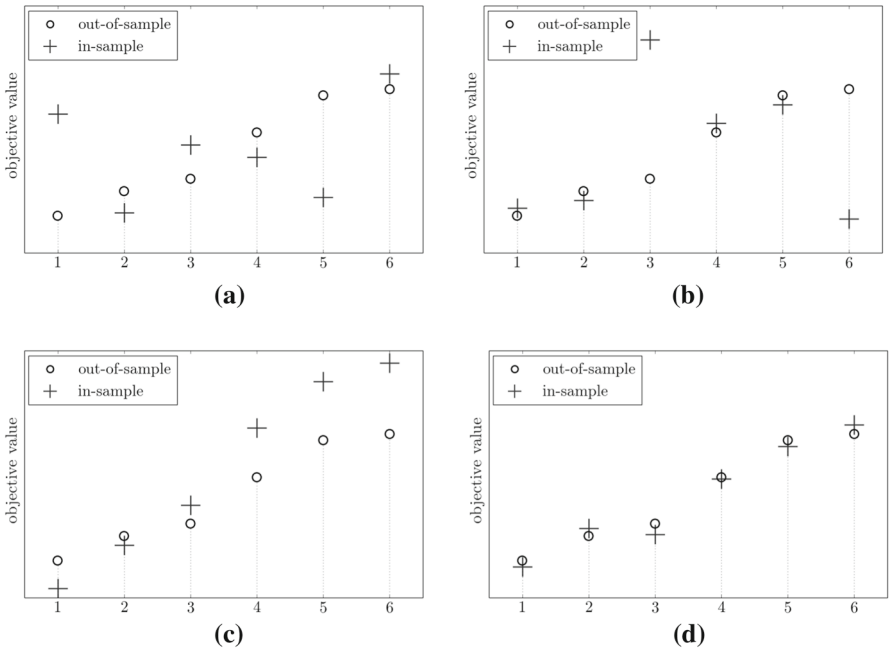


Fig. 1 Demonstration of different scenario trees and their properties with respect to in-sample and out-of-sample performance

In 1c, we show a good ranking of solutions, but with the wrong approximated values. Such a tree would be good for the optimization process, which would correctly find the optimal solution. Its out-of-sample value can be determined afterwards.

In 1d, the in-sample values approximate the out-of-sample value reasonably well. That implies that the ranking is approximately correct. We see that solutions nos 3 and 2 are ranked incorrectly, but since their out-of-sample values are similar, that would not cause a big error even if we solved the maximization problem over the solutions $\{1, 2, 3\}$.

Note

In this paper, we introduce a whole framework for scenario generation. But even if some other method (mentioned in the introduction) is used, visualization of the tree performance as in Fig. 1 can offer a fast and intuitive way to assess how good a scenario tree is. In other words, we use just the first step of the proposed framework: we apply some heuristic to generate a pool of solutions for problem (1) and visualize their out-of-sample and in-sample performance for a given tree.

2.2 Loss function

We introduce a loss function to measure the discrepancy between the out-of-sample and in-sample performance of a pool of solutions. The formulation of the loss function

is derived from our requirements on a good scenario tree. The smaller the value of the function, the better the tree (in our view) for the subsequent optimization procedure. We define the loss function in the following way:

$$L(\mathcal{T}, \mathcal{A}) = \sum_{x_a \in \mathcal{A}} \left(z_1^a \left(1 + z_2 \mathbb{1}_{[f(x_a,\mathcal{T}) > f(x_a,\xi)]}\right) \left(f(x_a, \mathcal{T}) - f(x_a, \xi)\right)^2 \right) \quad (3)$$

where $\mathbb{1}_{[condition]}$ takes the value 1 if the *condition* is met, 0 otherwise. The loss function is fundamentally the weighted sum of squares between in-sample and out-of-sample values that captures a basic fit between them (Requirement 4). A good fit between the in-sample and out-of-sample values implies that the ranking is more or less correct (Requirement 1).

Each square is further weighted with the term $(1 + z_2 \mathbb{1}_{[f(x_a,\mathcal{T}) > f(x_a,\xi)]})$, which penalizes approximations from above. Together with a potential high difference between in-sample and out-of-sample value, it penalizes the overconfident outliers (Requirement 2). If we dealt with a minimization problem, we would penalize the approximation from below as it leads to overconfident outliers.

Each term is also weighted with z_1^a which takes a higher value, the better (out-of-sample) a solution is. Thus, contrary to the previous weighting term, it does not depend on the scenario tree. The weights z_1^a put more emphasis on higher-quality solutions, for which it is more crucial that the scenario tree approximates better the underlying distribution.

Loss function (3) is formulated generally by using weights z_1^a and z_2. The weights are considered as parameters of the overall procedure, and they allow us to adjust the loss function based on problem specifications such as the heuristic for solution generation and the loss function minimization procedure (see the following section). For instance, if the heuristic procedure at times produces some bad solutions, z_1^a should be more progressive (adding more weights) towards better solutions compared to the case where all solutions from the heuristic phase are relatively good.

2.3 Minimization of the loss function

Our aim is to construct the scenario tree. Since we defined our measurement of discrepancy between the tree and the underlying distribution, we will naturally look for a tree, for which the discrepancy is as small as possible for a given pool of solutions.

When solving the optimization program (2), the set of scenarios (realizations of a random vector) and associated weights p enter the program as input data and we look for correct decisions. But for this task—minimization of the loss function—the roles are swapped. Scenarios and their weights p are free variables to be set, whereas decision variables of the optimization problem (heuristic solutions) are fixed and treated as input data.

Specifications of the minimization procedure depend on requirements of the scenario tree and the optimization problem we solve. For instance, it is possible that an application requires equiprobable scenarios. In such a case, weights p are no longer free variables but parameters in the in-sample evaluation function. This phenomenon

appears and is discussed also in Høyland and Wallace (2001). In addition, equiprobable scenarios should allow faster minimization of the loss function as it removes some non-convexities. On the other hand, we than need more scenarios to reach an equally good fit. Thus, the subsequent optimization will take more time. Hence, it is an open question whether it pays off (overall) or not. When using both scenarios and their weights p as free variables, we deal (most likely) with a non-convex problem, but we should be able to reach a better fit (the lower value of the loss function) between in-sample and out-of-sample performance—which is our main goal—despite the fact that the scenario tree does not form a properly defind probability distribution. This property is observed in the example in 2.3.1 or in Sect. 3 (Numerical analysis).

In Sect. 2.4, we discuss what happens if we have an integrality requirement on the scenarios. This makes the problem constrained. Similarly, we would get a constrained (non-linear and non-convex) problem if we let p be free variables, but require their sum to be 1 to form a probability distribution. We expect the minimization of the loss function to be too difficult for such constraints to pay off in most of the cases. Thus, there should be a really good reason for such a requirement.

A setting of the minimization procedure also depends on the structure of the optimization problem. There are problems with simple recourse that are inherently one-stage decision-making processes under uncertainty. An example could be a classical portfolio selection problem, or tasks with penalization for not satisfying some requirement—in 2.3.1 we discuss the stochastic knapsack problem. A more realistic example is a vehicle routing problem where a penalty for a late arrival to a customer is paid. We propose a heuristic for such problems, which is used in the example in 2.3.1. The heuristic utilizes the fact that there is no second stage decision, thus it is straightforward to derive the impact of a marginal change in scenarios and their weights p on the value of the loss function (expressed by the sub-gradient of the loss function with respect to scenarios and weights p).

In the case there is a decision to be made in the second stage that depends on the realization of randomness, one needs to consider the possible change of the solution. The results from parametric programming (sensitive analysis) may guide an analysis of the impact of the randomness on the objective value, and hence, on the loss function.

In other cases, for example integer (binary) scenarios (Sect. 2.4), a small change in scenarios does not have to cause any change in the loss function. Thus, a heuristic based on gradients is meaningless. But quite likely a different iterative (meta-)heuristic paradigm (for example genetic search) can be applied as the main (necessary) condition—fast evaluation of the function to be minimized—is satisfied. The evaluation of the loss function (3) is usually fairly fast because: the list of out-of-sample values is fixed (it needs to be evaluated only in the beginning of the procedure); the number of in-sample scenarios is (usually) small, so calculating the in-sample values should not take much time (unless we deal with a problem with a very difficult second stage); the remainder of the loss function consists of basic operations.

Deriving a list of all possible properties of the loss function minimization procedure (convexity, differentiability, types of constraints, etc.) for different classes of optimization problems goes beyond the scope of this paper. But we point out several phenomena associated with this phase.

The computational time spent on this phase should also be considered. As outlined, there is no guarantee that a better fit reached on the pool of solutions ensures also a better fit on the entire search space $\mathcal{X}$. Thus, it might be unproductive to spend excessively much time on searching for the very best solution of the loss function minimization. Dedicating more computational time to the choice of solutions (generating larger and higher-quality pool) and satisfying with a "good enough" fit in the second step might produce a higher-quality tree.

Note

In most applications, there will be a given number of scenarios based on how many of them can be computationally handled by the subsequent optimization procedure, for instance within a certain time limit. Then, the presented approach returns scenarios with the most similar performance to the original distribution with respect to the optimization task. However, it is possible to use the same framework to find scenarios that ensure some predefined loss function value (in our definition). The problem is to set such a threshold given several parameters (weights z_1^a and z_2) and solve a constrained optimization (non-linear, non-convex) program.

2.3.1 Illustration on the Stochastic knapsack problem

We demonstrate the usage of our framework on the classical stochastic knapsack problem where the items have uncertain volumes.

This belongs to the problems with simple recourse. We choose this simple example in order to show how the impact of scenarios on the loss function can be utilized in searching for scenarios. But mainly, we demonstrate how the scenarios are shaped based on the "needs" of the optimization problem.

The aim is to find K scenarios to represent a true distribution, which is given by discrete observations (historical data). That means the uncertainty is in both cases represented by the set of scenarios $\mathcal{S}$, each scenario has its p_s. For the true distribution, p_s is the probability of each scenario, with $p_s = \frac{1}{|\mathcal{S}|}$.

Let $\mathcal{I}$ be the set of items, item i having a value c_i and size w_{si} in scenario s, with the knapsack having a capacity W. All items we want to choose must be picked in the first stage and if their total size exceeds the capacity of the knapsack, we pay a unit penalty of d in the second stage. The objective is to maximize the expected profit. We formulate the optimization problem for decision variables $x \in \{0, 1\}^n$, where $x_i = 1$ if item i is picked, 0 otherwise;

$$\max_{x \in \{0,1\}^n} \sum_{i \in \mathcal{I}} c_i x_i - d \sum_{s \in \mathcal{S}} p_s e_s \tag{4}$$

$$\text{s.t} \qquad e_s \geq \sum_{i \in \mathcal{I}} w_{si} x_i - W \qquad \forall s \in \mathcal{S} \tag{5}$$

$$e_s \geq 0 \qquad \forall s \in \mathcal{S} \tag{6}$$

where e_s denotes exceeded capacity of the knapsack in scenario s. The in-sample and out-of-sample evaluation (depending on scenarios we use) of a given solution $\hat{x}$ is straightforward and fast:

$$f(\hat{x}, p, w) = \sum_{i \in \mathcal{I}} c_i \hat{x}_i - d \sum_{s \in \mathcal{S}} p_s \left(\sum_{i \in \mathcal{I}} w_{si} \hat{x}_i - W \right)^+ \tag{7}$$

where X^+ takes value X if $X \geq 0$, 0 otherwise.

The optimization model (4)–(6) and the evaluation function (7) hold for the scenario tree as well as for the true distribution (historical data also form a scenario tree). To distinguish between the two in the following text, we denote the true distribution $\sim = \{\hat{p}, \hat{w}\}$ and the desired scenario tree, consisting of K scenarios, as $\mathcal{T} = \{p, w\}$.

Our main focus is on the minimization of the loss function, so we choose a very simple heuristic to generate a pool of solutions $\mathcal{A}$. In every solution from the pool, an item i is randomly picked with probability q.

Since the pool of solutions is not changing during the subsequent mi999nimization of the loss function, the out-of-sample evaluation is performed only once. Then the list of out-of-sample values, denoted V (V_a for each solution $a \in \mathcal{A}$), enters the loss function minimization procedure as input data. We deal with an unconstrained problem

$$\min_{p,w} L(p, w, \mathcal{A}) \tag{8}$$

where the loss function is

$$L(p, w, \mathcal{A}) = \sum_{x_a \in \mathcal{A}} \left(z_1^a \left(1 + z_2 \mathbb{1}_{[f(x_a, p, w) > V_a]}\right) \left(f(x_a, p, w) - V_a\right)^2 \right) \tag{9}$$

The loss function is non-linear and non-convex in decision variables p and w. That makes the problem difficult to solve due to the existence of many local minima. Thus, we propose a heuristic that explores the search space in an efficient way to find a high-quality solution.

The core of the heuristic is the sub-gradient method that works in an iterative manner. It is designed for solving non-linear convex problems, so we add some additional features to prevent being trapped in the first local optimum along the way, and continue in searching for a better-quality solution. A detailed description of the procedure is provided in "Appendix A".

2.3.2 Computational test

We create two pools of solutions. One pool, called a *training pool*, is used to minimize the loss function (9) (by the procedure described in "Appendix A") to obtain a scenario tree $\mathcal{T} = \{p, w\}$. The second pool, called a *testing pool*, is used for evaluation of the obtained tree. The testing pool serves as a proxy for the entire search space $\mathcal{X}$, and thus, is used for the demonstration of the performance of the scenario tree.

In the literature, usually only the relation (quality gap) between in-sample and out-of-sample optimal solutions is presented. But we see two reasons to use a larger pool of (non-optimal) solutions for evaluation of the scenario tree: (i) Our framework aims mainly to difficult problems (for example integer ones). Very often these problems cannot be solved exactly, but a heuristic must be applied. Thus, we want to achieve a good ranking of (good) solutions, not just correctly classify the optimal one.[6] (ii) We can visually asses the quality of the scenario tree by observing how well the in-sample and out-of-sample values fit together and how they preserve the correct ranking.

In real usage, the heuristic from the first phase generates substantially weaker solutions (unless it gets very lucky) than what is assumed to be achievable with the subsequent optimization (otherwise there is no need of searching for a tree and solving the optimization problem). To mimic this property, we exclude the best 10% of the solutions from the training pool.

In our computational experiment, we generate 1000 scenarios for 19 items to represent the true distribution $\hat{w} = \{w_{si}\}_{s\in\mathcal{S},i\in\mathcal{I}}$. Our aim is to find a scenario tree consisting of only 3 scenarios. We use 200 solutions in the training pool and 400 in the testing pool.

To demonstrate the advantage of problem consideration in the scenario generation process, we apply our method on two cases. In the first case (case 1), we choose a capacity of the knapsack such that approximately half of all scenarios satisfy

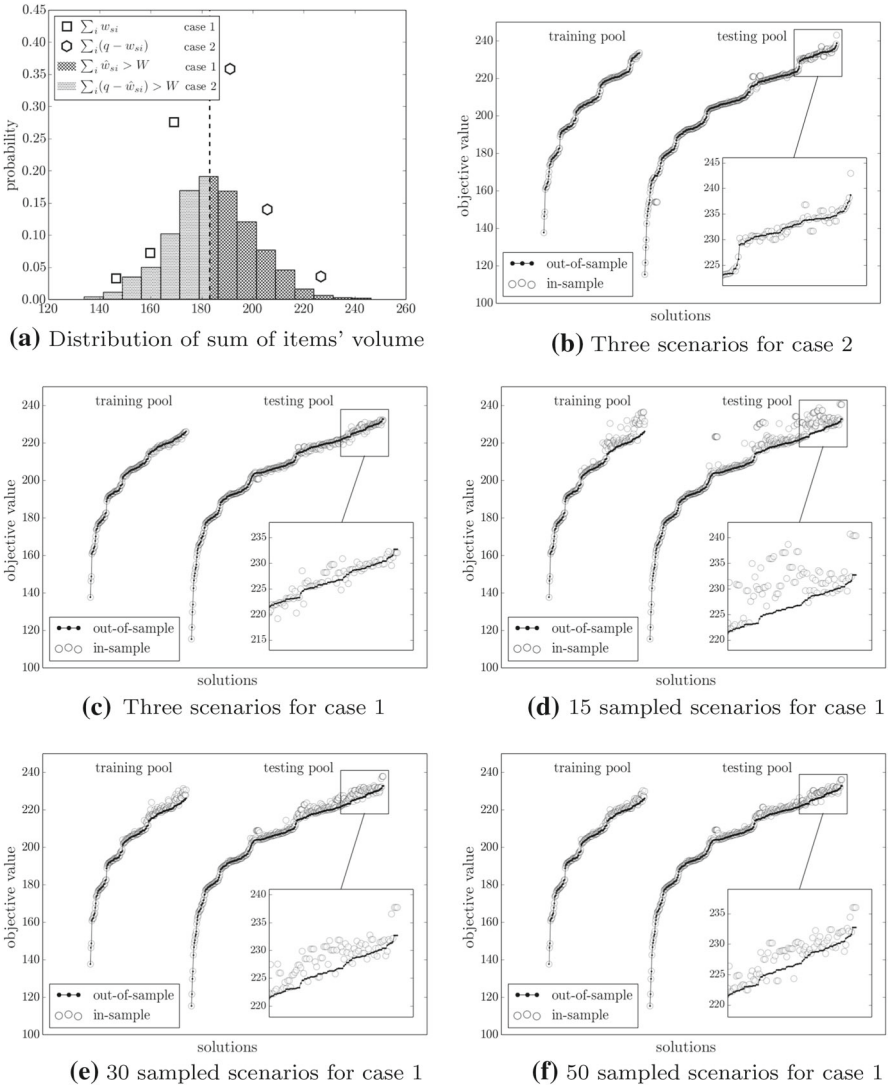
$$\sum_i \hat{w}_{si} > W. \tag{10}$$

In other words, only half of the scenarios may lead to a penalty. The rest of the scenarios do not have to be taken into account, since they do not lead to a penalty even for the decision $x = (1, 1, \ldots, 1)$ (all items are picked).

In the second case (case 2), we transform the original scenarios $\hat{w}^B_{si} = q - \hat{w}_{si}$, where $q > \max_{si} \hat{w}_{si}$. That is, small items become large and large items become small. We also set a new capacity of the knapsack W^B such that those scenarios, for which the condition (10) is not met, satisfy $\sum_i \hat{w}^B_{si} > W^B$. In other words, all the scenarios that may lead to a penalty in case 1 can be ignored in case 2. And vice versa, scenarios that may lead to a penalty in case 2, can be ignored in case 1.

We present numerical results for these two cases in Fig. 2. In 2a, we show the sum $\sum_i \hat{w}_{si}$ for all scenarios. We highlight the capacity of the knapsack W, which divides the scenarios into two sets—those that may lead to a penalty and those that never do (those can be ignored) in case 1, and oppositely in case 2. We shall see that this property was "discovered" and exploited by our framework and all scenarios were set such that they may lead to a penalty for some solution. We did not have to incorporate such a rule explicitly. It comes from the simple fact that more scenarios enable a better fit (lower value of the loss function). Thus, the minimization procedure, if designed properly, should use a maximal number of scenarios and not place any of them into the region that never leads to a penalty. This is a numerical counter-part of the analytical results in Fairbrother et al. (2017).

[6] Moreover, a correct in-tree classification of the optimal solution can be reached by luck for a random scenario tree. Thus, our computational test would have to be performed multiple times.



(a) Distribution of sum of items' volume

(b) Three scenarios for case 2

(c) Three scenarios for case 1

(d) 15 sampled scenarios for case 1

(e) 30 sampled scenarios for case 1

(f) 50 sampled scenarios for case 1

Fig. 2 Numerical results for a stochastic knapsack problem—comparison of three scenarios obtained by our framework and scenarios sampled from the original distribution

In 2b (case 1) and 2c (case 2), we show the discrepancy between in-sample and out-of-sample performance of the training and testing pool when the three final scenarios are used for in-sample evaluation. We zoom in on the best solutions from the testing pool where our main focus is. For comparison, we show Fig. 2d– 2f results for 15, 30 and 50 scenarios if they are randomly drawn from the original distribution. We show the results when the seed for the random draw is 1, so we were not cherry-picking some specific output. We can find much lower, as well as much higher, discrepancy if we choose different samples, especially in the case of smaller number (15) of scenarios.

The main point of this test is to demonstrate how our framework can shape the scenarios according to needs of the optimization problem, but without an explicit analysis of the problem. With such an approach we can tremendously reduce the number of scenarios. In our numerical example, three scenarios perform similarly[7] as fifty randomly picked scenarios.

Let us point out that any method for scenario generation, which is based on a good fit between the scenario tree and the original distribution without considering the optimization problem, would inevitably return an identical scenario tree for both cases, since the original distribution is shared in both cases. Therefore, a tree with three scenarios would generate (at best) two useful scenarios for one case and only one useful scenario for the other case. Obviously, such a scenario tree would perform worse than our tree in both cases, especially in the case where only one scenario is useful.

To put it from a different perspective, one scenario tree used for both cases would need at least twice as many scenarios as our tailor-made trees for each case to reach similar quality of performance. We admit that the problem is artificially set, but it clearly demonstrates our point that scenario trees derived purely from the original distribution, without considering the optimization problem, may lead to some redundant, or little important, scenarios.

Then, a natural question is how much it actually matters from a computational point of view to keep the number of scenarios small. That will be discussed in Sect. 4, where we discuss potential applications of our approach.

2.4 Binary distributions

In this section, we discuss problems where the uncertainty is described by a multivariate Bernoulli distribution (binary distribution in short). This represents a large class of real-world applications: a customer is present or not during delivery services, weather allows a ship to sail a certain edge or not, a machine is broken or not in a scheduling problem, just to give some examples.

And yet, stochastic programs with binary distributions are rarely studied in the literature and if they are [for example Ball et al. (1995) in the context of network reliability or Bent and Van Hentenryck (2004) in a routing problem], the focus is on the problem as such, not on handling scenarios (generation, reduction etc.). An exception is a paper Prochazka and Wallace (2018), where two useful methods are proposed; one for an efficient out-of-sample evaluation, and the second for reduction of the scenario tree into a minimal number of scenarios needed for an exact solution of the problem. However, to the best of our knowledge, there is no paper offering an efficient method for scenario generation for binary distributions (other than sampling) that would be suitable for solving, approximately, larger instances.

The advantage of our framework is that it does not rely on statistical properties and relationships among distributions. All we require are a heuristic for generating solutions, out-of-sample evaluations and an efficient procedure for minimizing the

[7] Based on a visual assessment. In order to provide a numerical justification of that claim, we would need to define an appropriate metric for the comparison.

loss function. The requirements on a good scenario tree and, therefore, the definition of the loss function, do not have to be adjusted.

Although the overall framework remains unchanged, we identify two cases of problems that differ in the procedure of loss function minimization. Let us consider the following example of the knapsack problem, where items have constant value and volume, but it is uncertain whether a particular item will appear or not after the decisions (to pick or not) are made. The optimization program (using the same notation as in 2.3.1) is as follows:

$$\max_{x\in\{0,1\}^n} \sum_{s\in\mathcal{S}} p_s \left(\sum_{i\in\mathcal{I}} r_{si} c_i x_i - d\ e_s \right) \tag{11}$$

$$\text{s.t} \quad e_s \geq \sum_{i\in\mathcal{I}} r_{si} w_i x_i - W \qquad \forall s \in \mathcal{S} \tag{12}$$

$$e_s \geq 0 \qquad \forall s \in \mathcal{S} \tag{13}$$

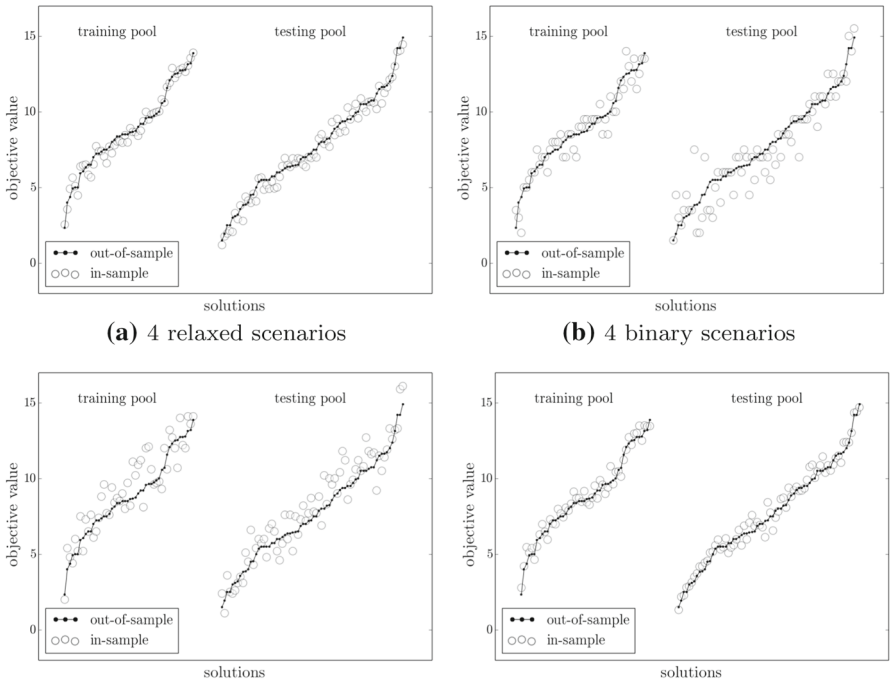
where r_{si} is the indicator of appearance of items, taking the value 1 if item i is present in scenario s, 0 if not. Assuming we have a pool of solutions, we formulate the loss function (3), where scenarios r_{si} and p_s are decision variables. For n items, we get 2^n possible combinations (scenarios), which results in unsolvable problems for large n.

Even though the scenarios are inherently binary, there is no reason to follow that restriction when constructing the scenario tree. Relaxation of scenarios (allowing all values for r_{si}) decreases the value of the loss function by extending the search space, and therefore, decreases discrepancy between performance of the scenario tree and the true distribution. Moreover, we can find sub-gradients with respect to r and p and use the same procedure (Algorithm 1) for minimizing the loss function as in the case of continuous scenarios.

Even though the optimization problem (11)–(13) gets a different interpretation: suddenly an item can be half-present and half-missing, it remains computationally meaningful. We need to remember that the goal is to find solutions for such a modified problem which are good also in the original problem.

In the following computational test, we consider an example with 8 items, each with a probability of appearance of 50% (independently of each other), so all the scenarios have equal probabilities. In total, there are 256 scenarios. We have a training pool consisting of 50 solutions and a testing pool of 70 solutions (all randomly generated). We want to have 4 scenarios to represent the whole distribution.

In Fig. 3a, we show the performance of 4 scenarios obtained by our framework when allowing relaxed scenarios. In 3b, we choose the best combination (with minimal loss function on the training pool) of 4 binary scenarios. We see that the relaxed scenarios perform much better on the testing pool in preserving ranking among solutions and approximating their true value (they perform better on the training pool by definition). We compare the performance with scenarios that are randomly sampled (random seed 1). Four relaxed scenarios obtained by our framework outperforms 20 sampled scenarios and provide comparable results as 50 sampled scenarios by visual assessment.



(a) 4 relaxed scenarios **(b)** 4 binary scenarios

(c) 20 sampled scenarios **(d)** 50 sampled scenarios

Fig. 3 Computational test for a problem with binary scenarios

There are, however, cases where the relaxation of scenarios is not helpful, albeit possible. An example would be a location-routing problem with uncertain customer appearance. A first-stage decision is the location of a warehouse, from which customers will be served (second-stage decision). It is uncertain which customers will use the service (contracts are not signed yet). Performing an optimal routing between customers in the second stage requires solving an optimization model that contains a constraint of type

$$\sum_i x_{ij} \geq \hat{d}_{sj} \tag{14}$$

where x_{ij} is a binary decision variable about a choice of traversing an arc from a node i to j. $\hat{d}_{sj}$ indicates whether a customer is present in node j in scenario s. In words, if a customer is present, he must be served. Let us assume the constraint is associated with a penalty for violation.

Any entry between 0 and 1 performs as if it was 1 in this constraint (at least one arc to j must be used). Let us assume that the parameter $\hat{d}_{sj}$ does not appear anywhere else in the optimization problem, just in (14). Then, whether $\hat{d}_{sj}$ is 1 or any arbitrary number between 0 and 1 does not have any impact on violation/non-violation of the constraint, and hence on the value of the objective function, and therefore on the value

of the loss function. As a consequence, there is no change in the value of the loss function if the parameter $\hat{d}_{sj}$, which is already greater than zero, is changed by an infinitesimally small number. Therefore, there is no useful change of (sub)gradients in the loss function minimization procedure and the relaxation of the scenarios is pointless.

So even if the scenarios can be principally continuous, there is no advantage in considering them to be so. We would still have to treat them as binaries ($\hat{d}_{sj}$ is either zero or greater than zero). That means that the minimization of the loss function is more problematic as it leads to solving a non-linear and non-convex problem with binary variables. In such a case, we suggest using meta-heuristics (for example genetic search), which are state-of-art methods for combinatorial optimization problems. But the general scheme—we minimize the discrepancy between in-sample and out-of-sample performance—remains unchanged.

Dealing with a network with n potential customers, the total number of possible scenarios is exponential in n. That is, again, often impossible to handle for large instances. Thus we can use only a subset of all possible scenarios. Our framework can help us identify a suitable set.

2.5 Feasibility

So far in the text, we assumed (relatively) complete recourse. That is, we assumed that all feasible first-stage decisions would lead to feasible second-stage problems for all possible values of the random variables. We also assumed that the heuristic producing our pool of solutions makes sure that they are all first-stage feasible.

In this section, we discuss the case where, for some reason, feasibility of the second-stage problem is an issue. We want to point out that this should be rather seldom. A constraint saying that (random) demand must always be satisfied leads to a worst-case (and at the same time, most likely, a subjective) model, which hardly makes sense. It may be true that a time window is hard in the sense that outside the time window it is impossible to deliver, but it hardly means that life does not go on if the truck is late. Rather, a penalty is incurred, and the activities continue. So, really hard constraints (violating the ideas behind relatively complete recourse) are extremely rare from an applied perspective. Even so, we shall discuss the issue to some extent here. Note that even in cases where some constraints need to be satisfied in every scenario, it is possible to use a penalty several orders of magnitude larger than the other profits (costs) in the objective function. Then, if there is a solution that can satisfy all the constraints, the optimization model will prefer it. If there is not such a solution, we can see which constraints have been violated. Such information is more valuable for analyzing the problem than a simple report that there is no feasible solution for the problem. For further discussion on feasibility modeling, see King and Wallace (2012).

In addition, it is also challenging to incorporate a (in)feasibility classification into our framework from a computational point of view as we lose some properties of the loss function, mainly the utilization of sub-gradients. Thus, we recommend to use penalization whenever possible.

However, if feasibility in the second stage really is an issue, for example due to a constraint related to some laws of physics, and which therefore cannot be violated, we introduce a new requirement on a good scenario tree that stands above those formulated in Sect. 2.1. We postulate it as follows:

0. A good scenario tree *classifies feasibility* of solutions correctly. That is, true feasible solutions are also feasible in the tree. The same holds for infeasible solutions. But as with outliers, we find it acceptable if occasionally a solution that is feasible in reality, is classified as infeasible by the tree, especially if the solution is weak. On the other hand, we want to avoid cases where solutions that are infeasible in reality, become feasible and very good in the tree.

Let us assume that the heuristic generates both feasible and infeasible solutions. We construct a list F of feasibility indicators, that is $F_a = 1$ if x_a is feasible (out-of-sample), 0 otherwise. Further, let the function $u(x, \mathcal{T})$ return 1 if x is feasible in the tree and 0 if the solution is infeasible in the tree.

We keep all the terms from the loss function as they were defined previously to reflect Requirements 1–4, and we add some new terms to capture Requirement 0. The loss function is

$$L^F(\mathcal{T}, \mathcal{A}) = L(\mathcal{T}, A) + \sum_{x_a \in \mathcal{A}} \Big(z_3^a \, u(x_a, \mathcal{T})(1 - F_a) + z_4^a \, \big(1 - u(x_a, \mathcal{T})\big) F_a \Big) \tag{15}$$

We simply add the weight z_3^a if an out-of-sample infeasible solution is classified as feasible by the tree, and weight z_4^a if an out-of-sample feasible solution is classified as infeasible. Since the first case is more critical, penalties z_3^a are set higher than z_4^a. Similarly as for z_1^a, weights z_3^a and z_4^a also depend on the quality of the solution x_a. The higher the out-of-sample value is, the higher weights we set, especially in the case of z_3^a, so we penalize very good (high value of the objective function) but infeasible solutions that are incorrectly classified as feasible by the tree.

Naturally, it is challenging to set all the weights z_1–z_4 properly to create a perfect balance between classification of feasibility and approximation of the out-of-sample values. The balance should be derived from the usage of the model and assessment of importance of having feasible solutions.

In addition, it is also challenging to incorporate a (in)feasibility classification into our framework from a computational point of view due to the added classification terms that cause discontinuity of the loss function. Thus, we recommend to use penalization whenever possible. A way to deal with the minimization of the discontinuous loss function, which is used in the following computational test, is discussed in "Appendix A".

Example

Let us consider an alternative version of the stochastic knapsack problem

$$\max_{x \in \{0,1\}^n} \sum_{i \in \mathcal{I}} c_i x_i - d \sum_{s \in \mathcal{S}} p_s e_s \tag{16}$$

$$\text{s.t} \quad 0 \geq \sum_{i \in \mathcal{I}} w_{si} x_i - W^{\max} \qquad \forall s \in \mathcal{S} \tag{17}$$

$$e_s \geq \sum_{i \in \mathcal{I}} w_{si} x_i - W \qquad \forall s \in \mathcal{S} \tag{18}$$

$$e_s \geq 0 \qquad \forall s \in \mathcal{S} \tag{19}$$

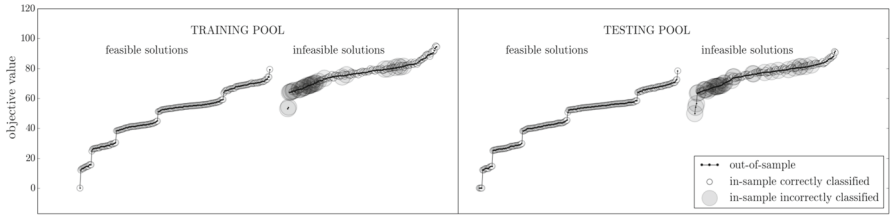
The total volume of picked items may exceed the capacity of the knapsack and we pay the corresponding penalty, but we cannot exceed the total value $W^{\max}$ ($> W$) in any single scenario.

We compare the performance of scenarios constructed by our framework with scenarios obtained by pure sampling from the original distribution and by using a copula-based heuristic (Kaut 2014) in Fig. 4. The comparison is made on the testing pool as it was not used to construct scenarios by our framework. The testing pool can be perceived as a proxy for the whole search space. Each pool consists of feasible and infeasible solutions sorted in ascending order based on the value of the objective function with highlighted incorrectly classified solutions.

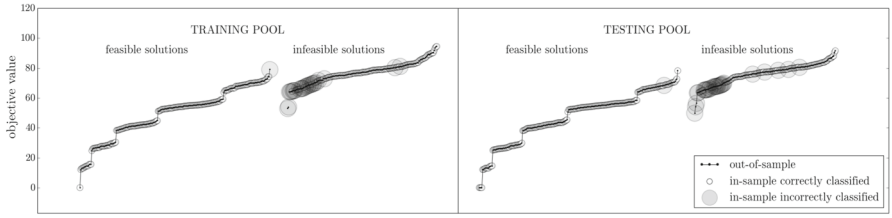
Since no feasible solution leads to a total size exceeding $W^{\max}$ in any realization (scenario) of the original distribution (by the definition of the feasible solution), the size also does not exceed $W^{\max}$ in any subset of scenarios drawn from the original distribution. In other words, sampled scenarios will always classify correctly out-of-sample feasible solutions, see Fig. 4a. However, it may happen that in some scenario from the original distribution, the limit $W^{\max}$ is exceeded, but such a scenario is not chosen in the sampled subset. Thus, several out-of-sample infeasible solutions are classified as feasible in the tree. Since some of them return high value of the objective function, they have a negative impact on the subsequent optimization as they provide too optimistic and almost always infeasible (out-of-sample) solutions, even if the sample is very large.

We observe a similar phenomena in the case the scenarios are obtained by the copula-based heuristic, Fig. 4b, c. The scenarios are constructed by generating realizations from marginal distributions and then combining them to match the shape of the copula of the original distribution. Thus, even feasible solutions can be occasionally incorrectly classified. This heuristic works extremely well when it comes to matching most of the properties of the original distributions (notice almost perfect estimation of the objective value). However, it is not designed to focus on capturing, in some sense, extreme scenarios that cause second-stage infeasibility. Thus, even a large scenario tree consisting of 100 scenarios might cause the occasional appearance of missclassified solutions that look feasible in the tree, but are infeasible in reality (due to one extreme scenario for example).

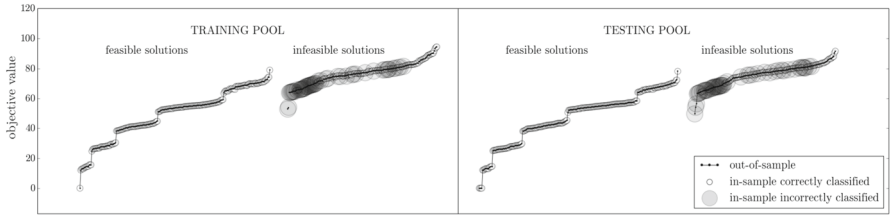
In contrast to the above described methods, our framework (Fig. 4d) primarily focuses on correct classification of (in)feasibility, especially on not letting good (high value of the objective function) infeasible solutions be classified as feasible in the tree. To satisfy that requirement, the tree tends to set scenarios more towards their extremes, and thus classify feasible solutions as infeasible more often in favor of correct classification of high-value infeasible solutions. That is a preferable situation for solving the optimization program.



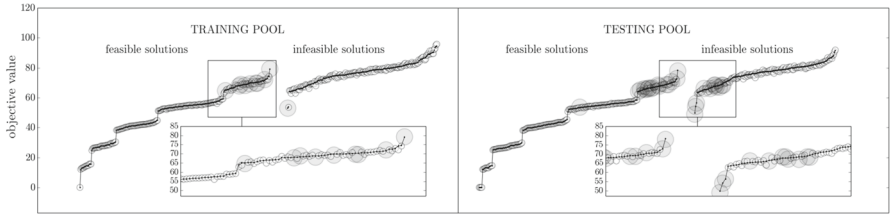
(a) 100 sampled scenarios



(b) 100 scenarios generated by copula-based heuristic



(c) 20 scenarios from copula heuristic



(d) 6 scenarios from our framework

Fig. 4 Computational test for a problem with infeasible solutions

Naturally, there is no guarantee that the presented approach is able to always capture the extreme directions of all scenarios to prevent that a good infeasible solution is incorrectly classified. This risk could be reduced by using a larger training pool.

To further minimize that risk, it would be possible to tighten constraints that cause infeasibility. In our case we could set $W^{\max}$ smaller, when solving the optimization program with the scenario tree. Then we have more certainty that we are on the "safe" side.

This leads to an idea that it is possible to set other parameters (input data), not only the scenarios, to mimic performance of the original distribution. Our approach

provides a framework that can serve that purpose (minimizing discrepancy between in-sample and out-of-sample performance). However, minimization of the loss function becomes more complicated as there are more variables to set.

3 Numerical analysis

In the previous sections, the computational experiments were performed on simple problems set up to illustrate some specific points. Here, we take a closer look at the computational aspects.

For the test, we use an extension of the model from Sect. 2.4 (random appearance of items) by considering several knapsacks. The computational time for solving this problem grows rapidly with the increasing number of scenarios and knapsacks, and thus, it is beneficial (or necessary) to have a small scenario tree. We compare the performance of scenarios produced by our framework and scenarios that are randomly sampled.

Let us denote by $\mathcal{J}$ the set of m identical knapsacks, each with capacity W. Thus, we need to decide not only which items i to pick, but also to which knapsack j each of them should be assigned. With notation from previous sections, the problem is formulated as follows:

$$\max_{x\in\{0,1\}^{m\times n}} \sum_{s\in\mathcal{S}} p_s \left[\sum_{j\in\mathcal{J}} \left(\sum_{i\in\mathcal{I}} r_{si} c_i x_{ij} \right) - d\, e_{js} \right] \tag{20}$$

$$\text{s.t} \quad \sum_{j\in\mathcal{J}} x_{ij} \leq 1 \qquad \forall i \in \mathcal{I} \tag{21}$$

$$e_{js} \geq \sum_{i\in\mathcal{I}} r_{si} w_i x_{ij} - W \forall j \in \mathcal{J} \qquad \forall s \in \mathcal{S} \tag{22}$$

$$e_{js} \geq 0 \qquad \forall j \in \mathcal{J} \quad \forall s \in \mathcal{S} \tag{23}$$

To perform our numerical analysis, we create several instances of the problem. We consider $\{2, 3, 4, 5\}$ knapsacks with the number of items being equal to $\{6, 8, 10\}$ times of the number of knapsacks. Input data are created randomly. Costs are taken from the uniform distribution $\mathcal{U}\{100, 140\}$ and weights from $\mathcal{U}\{7, 13\}$. We generate 2000 scenarios to represent our true distribution. An indicator r_{si} is 1 with probability 0.5. Each p_s is chosen randomly from $\mathcal{U}\{0, 1\}$ with subsequent normalization of the vector p (sum to be 1).

The capacity of the knapsack is set to 15 and the penalty to 14. These numbers are chosen such that the optimal (or best found) solution does not include all the items, but at the same time does not lead to a solution where the capacity is never exceeded (which is the case if the penalty is too high).

We create five instances for each $(|\mathcal{J}|, |\mathcal{I}|)$ pair and each instance is completely[8] re-run 10 times both by our framework and by sampling. We then display (in Table 1)

[8] That is, we start the whole procedure by generating a new pool of solutions with the subsequent loss function minimization and final optimization. In the case of sampling, we generate a new scenario tree.

the average statistics over these 50 runs on a personal laptop (Intel i5-4300 CPU, 4 x 4GB RAM, 1.6 GHz). The model is implemented in Python with linked Gurobi solver.

The heuristic to generate the pool is using the Gurobi solver and its embedded heuristic procedures. First, two scenarios are randomly sampled from the true distribution and the problem is solved with a short execution time limit (0.1s). All discovered feasible solutions from that short period of time are considered, evaluated out-of-sample and inserted into the pool. This whole procedure is repeated until the pool consists of at least 1000 solutions. We denote by h_{1000} the highest out-of-sample value observed in the pool.

We utilize the relaxed scenarios (see Sect. 2.4). The minimization of the loss function is then performed by the stochastic gradient descent (SGD) with multiple starts—in our case five. Each start is initialized with random scenarios taken from $\mathcal{U}\{0, 1\}$ and weights $p_s = \frac{1}{|\mathcal{S}|}$ for each s and run for 400 epochs. Then, additional 5000 epochs are run for the thread with the best loss function value reached after the 400 runs.

We display the CPU time spent on the minimization of the loss function in Table 1 (column *CPU scenarios*). However, in rows showing the pool's best solution, we show the CPU time needed to generate the entire pool in this column.

The most important statistics, is obviously the out-of-sample value, that is, the true value of the solution found by solving the optimization model (20)–(23) with corresponding scenarios. In addition to that, we show the gap between the out-of-sample (OOS) value and the in-sample (IS) value defined as:

$$\text{in-sample gap} = \frac{|\text{OOS} - \text{IS}|}{\text{OOS}} \tag{24}$$

We compare results obtained by using scenarios from our framework and scenarios that are randomly sampled from the real distribution (after normalization of the p vector). We consider 10, 20, 30 and 50 scenarios used in the tree. We observe the fast growth of the solution time needed, so we set a time limit of 120 seconds[9] on solving the problem. If the optimal solution is not returned, the best found solution is considered. Again, the main focus is on the out-of-sample value and the in-sample gap.

There are several takeaways from Table 1. The tree that consists of only 2 scenarios produced by our framework leads to a better solution than the best solution from the heuristic phase at all instances except for the smallest one—(2, 12).[10]

Notice how the comparison of the pool's best solution, our framework and sampling progresses with the increasing complexity of the problem (more knapsacks). For example, in the (2, 16) instance, even h_{1000} outperformed sampled trees consisting of 10 and 20 scenarios. Then, our framework was capable of outperforming all sampled scenario trees (even with 30 and 50 scenarios). In the (3, 24) case, h_{1000} was

[9] The time limit is set in wall-clock time. This corresponds to CPU time which oscillates around 430s due to presence of 4 cores (not all the subprocesses can be parallelized).

[10] In the (2, 12) case, there are only $2^{11} = 2048$ unique feasible solutions. Thus, it is not so surprising that some very high-quality (if not optimal) solution can be found among 1000 solutions, especially if they are not generated purely randomly.

Table 1 Comparison of our framework with randomly sampled scenarios

Instance	our framework					Sampling			
($\|J\|$, $\|I\|$)	# Tree	Out-of-sample	In-sample gap (%)	$\sum_s p_s$	CPU (s) scenarios	# Tree	Out-of-sample	in-sample gap (%)	CPU (s) optimization
(2, 12)	h_{1000}	349.2	–	–	3.5	10	338.3	13.5	0.1
	2	347.4	0.9	0.90	25.7	20	344.1	9.1	0.1
	4	348.9	0.9	0.92	26.5	30	344.7	6.7	0.1
						50	347.9	4.2	0.3
(2, 16)	h_{1000}	408.1	–	–	3.4	10	396.2	11.7	0.1
	2	413.1	1.8	0.93	25.8	20	404.7	6.6	0.1
	4	412.2	0.9	0.95	27.4	30	408.5	6.7	0.2
						50	411.5	4.2	0.3
(2, 20)	h_{1000}	379.1	–	–	4.4	10	362.2	14.1	0.1
	2	382.5	1.3	0.91	26.4	20	379.3	8.2	0.2
	4	382.9	0.7	0.93	27.8	30	379.8	7.2	0.3
						50	382.0	5.1	0.4
(3, 18)	h_{1000}	491.5	–	–	9.3	10	486.2	15.6	0.2
	2	496.5	2.2	0.89	28.3	20	493.2	10.5	0.6
	4	496.6	1.4	0.91	30.2	30	497.0	8.9	1.5
						50	498.5	6.7	4.7
(3, 24)	h_{1000}	529.3	–	–	10.2	10	525.3	13.9	0.3
	2	540.8	2.1	0.91	30.0	20	536.1	10.0	0.8
	4	538.8	2.0	0.93	57.4	30	538.3	8.2	1.6
						50	542.6	5.5	6.6
(3, 30)	h_{1000}	564.4	–	–	11.3	10	554.7	14.5	0.3
	2	572.5	2.2	0.91	31.2	20	574.4	10.1	0.9
	4	574.7	2.3	0.92	58.6	30	578.6	7.9	3.1
						50	577.5	5.9	10.8

Table 1 continued

Instance	our framework					Sampling			
($\|J\|$, $\|I\|$)	# Tree	Out-of-sample	In-sample gap (%)	$\sum_s p_s$	CPU (s) scenarios	# Tree	Out-of-sample	in-sample gap (%)	CPU (s) optimization
(4, 24)	h_{1000}	635.4	–	–	23.2	10	631.3	18.1	1.3
	2	638.5	2.0	0.88	32.3	20	640.8	13.1	9.2
	4	629.5	1.9	0.90	60.3	30	634.2	10.0	106.3
						50	625.4	7.1	389.3*
(4, 32)	h_{1000}	693.5	–	–	18.3	10	694.2	16.4	1.6
	2	715.2	1.6	0.90	34.6	20	702.4	11.7	15.2
	4	709.8	1.7	0.91	63.0	30	712.5	9.9	71.7
						50	708.7	7.4	376.1*
(4, 40)	h_{1000}	750.3	–	–	18.4	10	754.5	14.9	2.0
	2	758.9	1.5	0.90	63.7	20	767.9	10.2	11.4
	4	759.1	2.4	0.91	65.9	30	774.5	8.7	123.8
						50	777.2	6.8	405.9*
(5, 30)	h_{1000}	784.0	–	–	23.1	10	787.6	18.2	8.5
	2	798.6	1.6	0.84	63.9	20	804.4	13.5	239.2
	4	797.6	1.8	0.86	65.8	30	800.1	11.2	444.4*
						50	813.8	7.9	421.4*
(5, 40)	h_{1000}	861.1	–	–	22.7	10	905.5	16.2	14.2
	2	908.0	0.9	0.86	68.4	20	916.6	11.8	231.7
	4	902.9	1.1	0.86	70.5	30	922.4	9.7	438.3*
						50	919.6	7.6	431.2*
(5, 50)	h_{1000}	915.3	–	–	23.1	10	949.3	14.4	27.5
	2	954.5	1.3	0.87	73.3	20	969.6	11.4	231.6
	4	959.0	1.2	0.88	75.5	30	982.3	9.0	437.1*
						50	987.3	6.9	430.5*

*Some of the instances were cut off before finding the optimal solution

better "only" than 10 sampled scenarios, but further enhanced scenarios (fitted by the minimization of the loss function) produced better results than 20 and 30 sampled scenarios. In the most complex case—(5, 50)—the best solution from the pool was far behind the sampled tree of 10 scenarios, but after the enhancement, two scenarios performed better than 10 randomly sampled ones (but not better than larger trees).

Thus, for more complex cases, our framework produces relatively weaker solutions compared to sampling, but we contribute this fact mainly to a weaker heuristic phase. This leads to an idea for a future extension of the method—iteratively improving the pool by adding the solutions obtained from the current best tree (in the sense of the minimal loss function). After the addition of new solutions, the loss function can be recomputed and further minimized.

We do not observe any significant improvement by using 4 scenarios compared to 2 by our framework. We explain this counter-intuitive phenomena by the fact that no special adjustment of parameters (weights z_1 and z_2, learning rates, etc.), were used for the 4 scenario case, which might suit better for 2 scenarios.

An important (and positive) feature of our framework is the scalability. The time needed to set scenarios does not grow as fast as the computational time needed to solve the model.

Another point to mention is the smaller in-sample gap obtained by our framework. This is obviously not an issue if the problem we face was the final application. Then, applying the out-of-sample evaluation can follow to give the true value of the solution. However, in some applications, for example if the problem under study is a subproblem of a larger application, immediate knowledge (or a good approximation) of the out-of-sample value is very useful – for example in the case of decomposition techniques, where the value of the objective function leads to cuts generation for the first-stage problem. In the example of the stochastic knapsack problem, the average in-sample gap of the solution obtained by our framework varies around 1–2%, whereas sampling returns the gap of around 5–7% even for 50 scenarios (and more for a smaller number). Moreover, this is the average value—there is much larger variance[11] in this value when sampling is used compared to our approach.

Notice that letting p be free variables is utilized by our framework and the sum of the weights $\sum_s p_s$ is generally set below 1. This contributes to reaching a better in-sample fit as it decreases the natural overestimate of the objective value (present in the sampling case.)

Finally, we would like to emphasize that this is a "vanilla comparison". That is, we did not introduce any extra dependencies, nor other special attributes of the distribution. But as we demonstrate in Sect. 2.3.2 (identifying useless scenarios), our framework can be advantageous exactly in these "odd cases" compared to pure sampling. For example, we could come up with a true distribution and a problem, for which only one tenth (or any arbitrary fraction) of the mass is relevant. That would simply mean that we would need 10 times more sampled scenarios to produce similar results compared to our test. In other words, we could set the entire numerical test to be (arbitrarily much) in favor of our approach. We do not want to do that in this section, however, there are many real-life applications where the described structure appears—for example

[11] Based on our numerical test. Due to the lack of space, we do not report the variance in the table.

in some risk models, only the tail of the distribution may impact the decisions. Then, our framework can be even more effective tool than it appears in this test.

4 Applications

In this section, we summarize the main advantages and drawbacks of the framework. Based on that, we comment for what types of applications our approach could be beneficial, and where it is better to use a different method.

The main advantage is the number of scenarios needed to represent the underlying distribution. We demonstrate on several examples that just a small number of scenarios can perform as well as a much larger tree since they are "tailor-made" for a particular optimization problem. Our framework enables identifying spots where the scenarios are most useful. Thus, we aim for applications where it is crucial to keep the number of scenarios small.

The main disadvantage is the time to develop two subroutines; a heuristic for generating solutions and the subsequent procedure for minimization of the loss function, which is difficult. Therefore, we do not see any reason to use our framework for optimization problems that are run only once in principal (strategic problems) and/or can handle a large number of scenarios relatively easily (linear programs). In such a case, savings in computational time when solving the program (2) with a smaller number of scenarios do not exceed (most likely) the time needed for running the heuristic, tuning parameters of the loss function minimization procedure and running it. In addition, we need time to develop these procedures. Thus, for simple linear two-stage models, we recommend using some different method, for which a publicly released code can be found.

A typical example that could utilize our framework would be a stochastic vehicle routing problem[12] (VRP) that needs to be solved repeatedly. Stochasticity in routing problems can be related to uncertain travel conditions (potential congestion), uncertain demand, etc. See an overview of the field in Gendreau et al. (2016). For a dispatching company, this means solving the problem every day (or several times per day) with different input data, but the formulation of the optimization problem is the same. Thus, one can invest into the development of a heuristic, especially if it is likely that one needs the heuristic even for solving the final optimization problem. Typical state-of-art heuristics (genetic search, adaptive neighborhood search, simulated annealing, etc.) for VRPs produce many solutions very quickly during the process. We can imagine that the pool of solutions is obtained by multiple runs of the heuristic with different subset of scenarios (for example randomly sampled), and then, the same heuristic is run with the final scenario tree obtained by minimizing the loss function. We can even imagine that the two subroutines—solution search and the scenario tree search—can be merged and cleverly implemented into one heuristic with several updates of both subprocesses.

[12] Or generally a difficult problem, where the number of scenarios, and thus the number of constraints/variables significantly influence the computational time.

Another application of our framework is on problems with distributions that have no alternative ways to generate scenarios, except for sampling. But sampling often requires too many scenarios to provide a stable solution, and that is sometimes unaffordable from a computational point of view. To this class of problems we can include programs with the multivariate Bernoulli distribution. An example is VRPs with uncertain appearance of customers. We already made some points that play in favor of our approach when dealing with VRPs. Issues related to binary distributions are analyzed in Sect. 2.4.

There are many real-world application where we need to work with a combination of different distributions (different random variables). Some of them might be empirical, some theoretical, some might by binary, some continuous, etc. It is difficult to handle this issue by other methods for scenario generation. Due to the fact that our framework does not rely on statistical properties of the distribution, but simply looks for a scenario tree that mimics the out-of-sample performance in some (defined) way, it can, in principal, be used in such a case once we are able to evaluate the out-of-sample quality of a solution.

The last area we want to mention is multi-stage optimization [for example Dantzig and Infanger (1993)]. Our numerical experiments show that our framework can produce scenario trees with a smaller number of scenarios than other methods with the same level of solution quality. Since the size of the scenario tree growth exponentially with the number of stages in a multi-stage setting, the smaller number of scenarios per stage implies significant reduction in size of the overall tree. Moreover, the scenarios can be generated stage by stage by simply trying to mimic the behavior of the original distribution at that particular stage. Thus, we do not have to control dependencies across the stages as is the case when matching statistical properties of the scenario tree and the original distribution. This is not further studied in this paper.

5 Conclusion

We introduce a new problem-oriented approach for generating scenarios for stochastic optimization. It is not based on matching the scenario tree and the underlying distribution in some probabilistic sense, but it sets the scenario tree in such a way that the tree performs similarly as the original distribution. The performance is evaluated on a pool of solutions that are produced by some heuristics. The similarity of performance is measured by a loss function that we introduce. Its formulation is derived from our postulates on what constitutes a good scenario tree.

Hence, it is the optimization problem that drives the scenario generation by searching for a tree that minimizes the loss function. Thus, the parts of the original distributions that are more crucial for good solutions are approximated with greater emphasis. This approach often leads to a smaller number of scenarios compared with other methods. The main disadvantage is that two main subroutines—the heuristic for generating solutions and the procedure for minimization of the loss function—are not necessarily trivial and need to be developed specially for a given problem.

Thus, the framework is suited for applications where it is critical to use as few scenarios as possible, for example difficult problems (non-linear, integer) that seri-

ously increase their complexity with the number of used scenarios. We believe it is worthwhile to choose our approach especially in cases where the problem is solved repeatedly (with different input data), so we can utilize the implemented parts several times.

Another usage of our framework is on problems where uncertainty is represented by distributions, for which there is no alternative to pure sampling (even that might be problematic when we have a combination of different distribution types). In this paper, we discuss the case of binary distributions that have many applications in real life, for example a customer that appears randomly in a routing problem, a machine that might not work in a scheduling problem, or a cargo from A to B that is available on a future spot market with a certain probability.

We offer an alternative point of view on the relationship between optimization problems and scenario trees. Even if a different method for scenario generation is used, a simple visual test that compares in-sample and out-of-sample performance of a pool of heuristic solutions, can provide an intuitive way to assess the quality of the tree.

We introduce some ideas for further extension of this research. One is to use our framework on multi-stage optimization problems, where the number of scenarios grows exponentially with the number of stages. Thus, it is desirable to have as few scenarios per stage as possible. Another idea, that is not elaborated in this paper, is to consider other input data (parameters), and not only the scenarios, to be set according to our framework. Thus, we would create a modified problem, whose solution would, hopefully, be similar to the solutions of the original problem using the entire distribution.

Appendix A: Heuristic for loss function minimization

We provide a detailed description of the heuristic used in the example 2.3.1 for the minimization of the loss function. The method is based on sub-gradients of the loss function with respect to the decision variables, denoted g_p and g_w. In order to use them, we derive[13] $\frac{dL}{df}$—the gradient of the loss function with respect to the in-sample evaluation function, further $\frac{df}{de}$—the gradient of the in-sample evaluation function with respect to the exceeded capacity, and so on. By doing so, we get a chain of simple operations (square, multiplication, addition, max(),[14] etc.), for which we have standard differentiation rules. By simple applications of the chain rule we get the desired sub-gradients g_p and g_w.

Let us note that in other optimization problems it might not be so straightforward to derive sub-gradients as in our case. More complicated functions might come into play. Thus it could require more effort in analytical derivation or the use of numerical sub-

[13] We can compute the numerical approximation of sub-gradients directly from the definition of the sub-gradient instead of deriving them analytically. That is, however, computationally too expensive and significantly influence the computational time.

[14] The function $\max(\cdot, \cdot)$ is non-differentiable at the point where the two arguments are equal. Hence we use sub-gradient instead of gradient to be precise. It has no practical impact from the computational point of view.

gradients, which should always be available, at least in in principle. Or it is possible to choose a different method from non-linear optimization theory, that is not based on gradients [see some textbook on non-linear optimization, for instance Hendrix and Tóth (2010), Boyd and Vandenberghe (2004)].

With the sub-gradient method, we take a small step in the direction of the negative sub-gradient, that is, in the direction of the steepest descent, at every iteration. Such an approach converges to a local minimum, which is also a global minimum in the case of convex minimization. However, not in our case, so we add two features to enhance exploration of the search space in order to avoid termination of our procedure at some low-quality local minimum.

The first feature is to use multiple starts of the procedure from different initial points. The second feature is the recognition and replacement of "useless" scenarios. We recognize these scenarios by evaluating their impact on the loss function. The impact is defined as the change of the loss function values if we remove a particular scenario from the scenario tree. If the change is very small, it means that the particular scenario is not very useful, and we replace the scenario by a new one (chosen randomly). The heuristic is summarized in Algorithm 1.

Algorithm 1 Minimization of the loss function

1: **for** $m = 1$ to M **do**
2: **initialize** p and w
3: **for** $j = 1$ to J **do**
4: **compute** sub-gradients g_p and g_w
5: $p = p - \alpha_j^p g_p$
6: $w = w - \alpha_j^w g_w$
7: **compute** $L_{jm}(p, w)$ according to (9)
8: **for** $k = 1$ to K **do**
9: **if** $\|g_{w_k}\|_\infty < \epsilon^w$ **then**
10: $\tilde{p} = \{p_l : l \neq k\}$
11: $\tilde{w} = \{w_l : l \neq k\}$
12: **if** $|L(\tilde{p}, \tilde{w}) - L(p, w)| < \epsilon^L$ **then**
13: **replace** w_l with a new scenario
14: **choose** p and w that correspond to the smallest L_{jm} value.

The parameters p and w are updated (rows 5 and 6 in Algorithm 1) by small steps in the direction of the steepest descent. The step sizes, denoted α^p and α^w, are decreased with the increasing number of iterations. They are hyper-parameters that enter the procedure and need to be carefully set (too small steps lead to slow convergence, too large to oscillation or divergence). We set these steps based on some trial tests.

The routine of scenario usefulness assessment is computationally expensive. It would require computation of the loss function K times at every iteration. To avoid it, we run a pre-test, where we check the ∞-norm of the sub-gradient related to each scenario (row 9), which allows us to break the test once we find one of its element greater than ϵ^w. Only if the sub-gradient is small, do we proceed to the evaluation of the impact on the loss function.

Initialization and replacement of scenarios is performed by a random draw from the original data $\hat{w}$, weights p are randomly set. Algorithm 1 is just a pseudo-code, not the most efficient implementation. Obviously, storing all L_{jm} is not necessary, since at any time, we can store just two best values—a local one for the j cycle and a global one for the m cycle. We can also compute the value of the loss function while evaluating the sub-gradients of the function.

Note

We do not claim that this is the most effective heuristic for the problem. Most likely it is not. It is based on a simple sub-gradient method. If the main focus was on developing the most efficient algorithm for this task, it would be possible to build it on more sophisticated algorithms for non-linear optimization, such as the adaptive gradient method, possibly with momentum, or methods based on the second sub-derivative.

We believe this can still serve as an inspiration for developing more efficient algorithms, if needed. We pointed out some issues and suggestions how to overcome them. But for some applications, the presented algorithm is "good enough", as it was in our case. It is important to realize that even if we could guarantee the global optimum of the loss function, the whole framework would still be a heuristic in the sense that there are no guarantees relative to other feasible solutions that are not included in the pool.

Heuristic for (in)feasibility classification

The main issue when considering (in)feasibility is the minimization of the loss function (15). The problem is that we cannot utilize the sub-gradients of the function $u(x, \mathcal{T})$. The function is not continuous and returns only two values, that means its derivative is 0 (if it is defined). One needs to either use methods for non-linear optimization that do not utilize derivatives or approximate the function u with some differentiable function. The latter approach is used for the computational test in Sect. 2.5.

In our computational test, the classification of (in)feasibility, i.e., the approximation of $u(x, \mathcal{T})$, is performed by a simple neural network model with one hidden layer, sigmoid function as the activation function in both layers and the sum of squares as the measurement of the error. As input we use the vector e_s scaled to (0, 1). The neural network is trained on the training pool of heuristically obtained solutions. This simple model works well in our case and correctly classifies (in)feasibility of solutions.

The main advantage of this approach is that the neural network can back-propagate the sub-gradients of error (miss-classified solutions) via the weights of the neural network to the sub-gradient of the e_s vector and further to scenarios w_{si}. Thus, we can, in principal, use Algorithm 1 to find the scenario tree.

References

Ball MO, Colbourn CJ, Provan JS (1995) Network reliability. In: Ball MO, Magnanti TL, Monma CL, Nemhauser GL (eds) Network models, volume 7 of handbooks in operation research & management science, chapter 11. North-Holland, Amsterdam

Bent RW, Van Hentenryck P (2004) Scenario-based planning for partially dynamic vehicle routing with stochastic customers. Oper Res 52(6):977–987

Birge J, Louveaux F (1997) Introduction to stochastic programming. Springer, New York

Boyd S, Vandenberghe L (2004) Convex optimization. Cambridge University Press, Cambridge

Cario MC, Nelson B (1997) Modeling and generating random vectors with arbitrary marginal distributions and correlation matrix. Technical report, Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, IL

Dantzig GB, Infanger G (1993) Multi-stage stochastic linear programs for portfolio optimization. Ann Oper Res 45(1):59–76

Fairbrother J, Turner A, Wallace S (2017) Problem-driven scenario generation: an analytical approach to stochastic programs with tail risk measure. ArXiv e-print 1511:03074

Garey M, Johnson D (1979) Computers and intractability, a guide to the theory of NP-completeness. Freeman, New York

Gendreau M, Jabali O, Rei W (2016) 50th anniversary invited article—future research directions in stochastic vehicle routing. Transp Sci 50(4):1163–1173

Haugland D, Wallace SW (1988) Solving many linear programs that differ only in the righthand side. Eur J Oper Res 37(3):318–324

Hendrix E, Tóth B (2010) Introduction to nonlinear and global optimization. Springer, New York

Higle JL, Sen S (1991) Stochastic decomposition: an algorithm for two-stage linear programs with recourse. Math Oper Res 16:650–669

Høyland K, Wallace SW (2001) Generating scenario trees for multistage decision problems. Manag Sci 47(2):295–307

Kall P, Wallace SW (1994) Stochastic programming. Wiley, Chichester

Kaut M (2014) A copula-based heuristic for scenario generation. Comput Manag Sci 11(4):503–516

Kaut M, Wallace SW (2007) Evaluation of scenario-generation methods for stochastic programming. Pac J Optim 3(2):257–271

Kaut M, Wallace SW, Vladimirou H, Zenios S (2007) Stability analysis of portfolio management with conditional value-at-risk. Quant Finance 7(4):397–409

King AJ, Wallace SW (2012) Modeling feasibility and dynamics, chapter 2. In: Modeling with stochastic programming. Springer series in operations research and financial engineering. Springer, New York

King AJ, Wallace SW, Kaut M (2012) Scenario-tree generation, chapter 4. In: Modeling with stochastic programming. Springer series in operations research and financial engineering. Springer, New York

Lurie PM, Goldberg MS (1998) An approximate method for sampling correlated random variables from partially-specified distributions. Manag Sci 44(2):203–218

Pflug GC (2001) Scenario tree generation for multiperiod financial optimization by optimal discretization. Math Program 89(2):251–271

Prochazka V, Wallace SW (2018) Stochastic programs with binary distributions: structural properties of scenario trees and algorithms. Comput Manag Sci 15(3):397–410