JuDGE.jl: a Julia package for optimizing capacity expansion

Anthony Downward
University of Auckland, a.downward@auckland.ac.nz,

Regan Baucke
Air New Zealand, regan.baucke@gmail.com,

Andrew B. Philpott
University of Auckland, a.philpott@auckland.ac.nz

We present JuDGE.jl, an open-source Julia package for solving multistage stochastic capacity expansion problems using Dantzig-Wolfe decomposition. Models for JuDGE.jl are built using JuMP, the algebraic modelling language in Julia, and solved by repeatedly applying mixed-integer programming. We illustrate JuDGE.jl by formulating and solving a toy knapsack problem, and demonstrate the performance of JuDGE.jl on problems of increasing size in comparison with a deterministic equivalent model.

Key words: stochastic programming, capacity planning, Dantzig-Wolfe decomposition, Julia, JuMP

1. Introduction

Multistage stochastic capacity expansion problems have been widely studied in operations research for many years. The survey paper by Luss (1982) describes the early history of deterministic models. For more recent accounts of stochastic capacity expansion models see e.g. Van Mieghem (2003), Snyder (2006), Govindan et al. (2017). Capacity expansion models arise in facility location (Snyder 2006), supply-chain network design (Ahmed et al. 2003), vaccination planning (Özaltın et al. 2018), and natural disaster planning (Mete and Zabinsky 2010, Hong et al. 2015). Our interest in these models is motivated by capacity expansion planning in electricity systems (see e.g. Borison et al. 1984, Flores-Quiroz et al. 2016, Villumsen et al. 2012, Moreno et al. 2017, Wu et al. 2017, Wogrin et al. 2020).

Solving instances of stochastic capacity expansion models as mathematical programs at realistic scale is extremely difficult (Ahmed et al. 2003) because of the presence of integer

variables. Stochastic integer programming problems inevitably become too large to solve as a single model, so decomposition is needed.

Various forms of Lagrangian decomposition have been proposed for stochastic integer programming (see e.g. Borison et al. 1984, Carøe and Schultz 1999, Dentcheva and Römisch 2004, Sen and Sherali 2006). As shown by Dentcheva and Römisch (2004) scenario decomposition leads to a tighter Lagrangian relaxation than other forms of decomposition such as by scenario-tree node (Borison et al. 1984). In recent years, the progressive hedging algorithm of Rockafellar and Wets (1991) has emerged as the most popular scenario decomposition approach for stochastic integer programming. A major contribution to its adoption has been the provision of reliable software implementations on parallel processors (see e.g. Watson and Woodruff 2011, Munoz and Watson 2015).

Scenario decomposition becomes difficult to use in capacity planning problems that must distinguish between stochastic investment decisions and stochastic operating decisions. These can be modeled using *multi-horizon* scenario trees (Kaut et al. 2014). Scenario decomposition that does not take advantage of this structure will give an enormous number of scenario subproblems, each of which will be a large-scale (potentially intractable) mixed-integer program.

An alternative approach to dealing with the scale of a stochastic capacity expansion model is to use Dantzig-Wolfe decomposition as described in Singh et al. (2009). This uses a nodal decomposition (yielding tractable subproblems) while taking advantage of the multi-horizon problem structure in order to yield a tight relaxation of the original model. This algorithm has been applied in a number of practical settings in the electricity sector (see e.g. Flores-Quiroz et al. 2016, Villumsen et al. 2012, Moreno et al. 2017, Wu et al. 2017, Wogrin et al. 2020), but has not received much attention in other settings.

One drawback of the Dantzig-Wolfe approach is the modeling overhead required to define a suitable master problem and subproblems, and deal with the interaction between these. This can deter even the most enthusiastic practitioners, who may find it easier to formulate a large-scale deterministic mixed integer program (MIP) in an appropriate modeling language, and to deliver this to a commercial solver.

Our goal in this paper is to describe JuDGE.jl (Julia Decomposition for Generalized Expansion), a Julia package for solving capacity-expansion problems formulated in the JuMP modeling language (Dunning et al. 2017), and to demonstrate its effectiveness on

some test problems. JuMP is a modeling language in Julia that provides a simple, yet powerful, syntax to describe optimization problems; by utilizing the MathOptInterface package, these problems can be solved by a number of open-source and commercial solvers. The convenience of JuMP and the speed of Julia code has proved to be an ideal environment for implementing decomposition methods for large-scale mathematical programming problems (see e.g. Dowson 2017). We hope that the JuDGE.jl package (henceforth JuDGE) will provide practitioners with a useful framework for building tractable capacity expansion planning models at scale.

The paper is laid out as follows. In the next section we describe the class of problems that JuDGE can solve. In section 3 we describe the components of the package and how they are used to construct and solve a problem instance. Section 4 then demonstrates JuDGE by applying the package step by step to an illustrative example based around a stochastic knapsack problem. Section 5 describes some of the advanced features of JuDGE that enable it to tackle quite general problems. We conclude the paper with the results of some computational experiments in which JuDGE is applied to instances of a stochastic optimization problem at increasing scale, and solution times are compared with those of deterministic equivalent MIPs.

2. Capacity planning problems

In this paper we are interested in solving stochastic optimization problems formulated in a *scenario tree* with nodes $n \in \mathcal{N}$ and leaves in $\mathcal{L}$. A pictorial representation of a scenario tree with four time-stages is given in Figure 1.

The probability of the state of the world represented by node n is denoted $\phi(n)$. By convention we number the root node $n = 1$. The unique predecessor of node $n \neq 1$ is denoted by n_-. We denote the set of children of node $n \in \mathcal{N} \setminus \mathcal{L}$ by n_+, and denote its cardinality by $|n_+|$. The set of predecessors of node n on the path from n to node 1 is denoted $\mathcal{P}_n$ (so $\mathcal{P}_n = \{n, n_-, n_{--}, \ldots, 1\}$), where we use the natural definitions for n_{--}. The set of successors of node n is $\mathcal{S}_n = \{n \cup n_+ \cup n_{++} \cup \ldots\}$ where n_{++} is defined in the obvious way. The depth $\delta(n)$ of node n is $|\mathcal{P}_n| - 1$, the number of arcs on the path to node n, so $\delta(1) = 0$. Moreover, if we assume that every leaf node has the same depth, say $\delta_{\mathcal{L}}$, then the depth of a node can be interpreted as a time index $t = 0, 1, 2, \ldots, T = \delta_{\mathcal{L}}$.

Given a scenario tree, JuDGE is designed to solve problems of the following form:

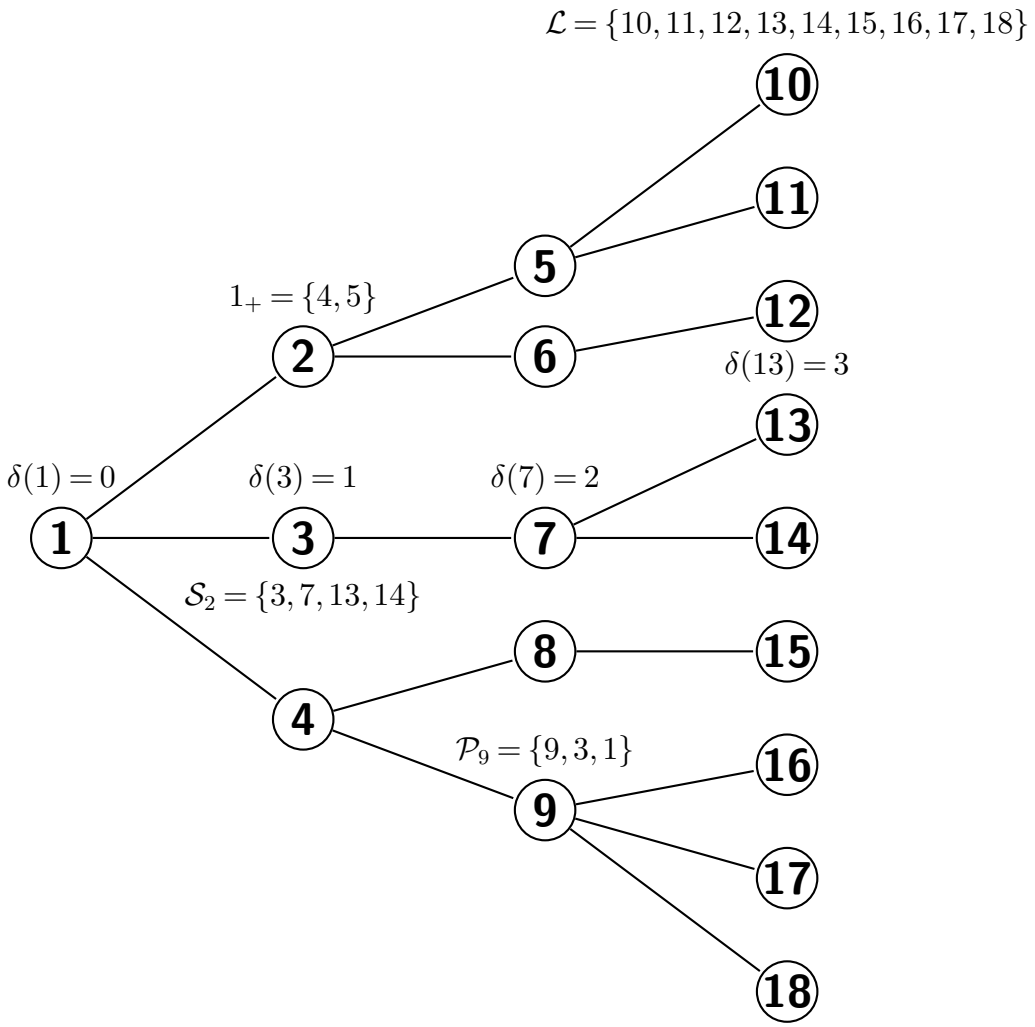

Figure 1 **A scenario tree with nodes $\mathcal{N} = \{1, 2, \ldots, 18\}$, and $T = 4$**

$$
\begin{aligned}
\mathrm{P}: \quad \min_{x,y} \quad & \sum_{n \in \mathcal{N}} \phi_n (c_n^\top x_n + q_n^\top y_n) \\
\text{s.t.} \quad & A_n y_n \leq u_n + U_n \sum_{h \in \mathcal{P}_n} x_h, \; \forall n \in \mathcal{N}, \\
& y_n \in \mathcal{Y}_n, \; \forall n \in \mathcal{N}, \\
& \sum_{h \in \mathcal{P}_n} x_h \leq \mathbf{1}^m, \; \forall n \in \mathcal{N}, \\
& x_n \in \{0, 1\}^m, \; \forall n \in \mathcal{N}.
\end{aligned}
$$

- m is the number of binary expansion variables;
- x_n is a vector of binary indicator variables for the expansions at node n;
- y_n is a vector of subproblem variables;

- $\mathcal{Y}_n$ is the subproblem feasibility set;
- u_n is a vector denoting the initial capacity available for each constraint corresponding to node n, if no capacity expansions have occurred;
- U_n is a matrix, both mapping expansions to constraints in the subproblems, and also giving the amount of expansion. This allows multiple expansions to apply to a single constraint, as well as a single expansion to apply to multiple constraints.

Observe that P constrains expansions so that for every scenario any type of expansion can be chosen at most once over the planning horizon. This is the SV1 form of capacity expansion model discussed in Singh et al. (2009). This restriction is not too onerous in practice; repeated capacity expansion of a certain type can be modeled by duplicating binary variables.

In typical applications $\mathcal{Y}_n$ will not be convex, for example some variables y_n might be constrained to be integer or binary. To obtain a (near) globally optimal solution to large instances of P might therefore be very difficult even with the best mixed-integer programming solvers. By exploiting the block structure of this problem using Dantzig-Wolfe decomposition, JuDGE enables us to solve very large instances of P.

The restricted master problem in this decomposition is as follows.

$$
\begin{aligned}
\text{RMP:} \quad \min_{w,x} \quad & \sum_{n\in\mathcal{N}} \phi_n c_n^\top x_n + \sum_{n\in\mathcal{N}} \sum_{j\in J_n} \phi_n \psi_n^j w_n^j \\
\text{s.t.} \quad & \sum_{j\in\mathcal{J}_n} \hat{z}_n^j w_n^j \le \sum_{h\in\mathcal{P}(n)} x_h, \quad n\in\mathcal{N}, && [\pi_n] \qquad (1)\\
& \sum_{j\in\mathcal{J}_n} w_n^j = 1, \quad n\in\mathcal{N}, && [\mu_n] \qquad (2)\\
& w_n^j \ge 0, \quad n\in\mathcal{N}, j\in\mathcal{J}_n, \\
& x_n \ge 0, \quad n\in\mathcal{N}.
\end{aligned}
$$

The optimal dual variables to (1) and (2) are then used to formulate a subproblem for each node $n\in\mathcal{N}$ as follows.

$$
\begin{aligned}
\text{SP}_n: \quad \min_{y,z} \quad & \phi_n q_n^\top y_n - \pi_n^\top z_n - \mu_n \\
\text{s.t.} \quad & A_n y_n \le u_n + U_n z_n \\
& y_n \in \mathcal{Y}_n, \\
& z_n \in \{0,1\}^m,
\end{aligned}
$$

where z_n is a vector specifying whether the expanded capacity (x_n) will be used for node n. For each node $n \in \mathcal{N}$, the solution $(\hat{y}_n^j, \hat{z}_n^j)$ to SP_n provides a column of 0's and 1's ($\hat{z}_n^j$) that is added to RMP with corresponding variable w_n^j. The cost coefficient ψ_n^j of this variable in RMP is the value $q_n^\top \hat{y}_n^j$. After each iteration the sets $\mathcal{J}_n$ are updated to be the set of columns produced so far for node n.

3. JuDGE

JuDGE is an open-source Julia package available for free download from GitHub. JuDGE can be installed via the following function call in the Julia REPL:

```
julia> ] add "https://github.com/reganbaucke/JuDGE.jl"
```

3.1. Implementation of capacity expansion problems in JuDGE

JuDGE works in the following way: JuDGE allows the user to define a scenario tree, in doing so defining $\mathcal{P}_n$ and ϕ_n, for each node in the tree. The user then describes to JuDGE the set of expansion variables and their costs using JuMP syntax (Dunning et al. 2017), and formulates the optimization problem SP_n for each node n. All problems are assumed by JuDGE to be minimization problems. Given the subproblem formulation, JuDGE automatically constructs a restricted master problem for the Dantzig-Wolfe decomposition, with the necessary links to the subproblems for each node.

When solving P, JuDGE automatically coordinates the dual information from the master problem (π_n and μ_n) into the objective functions of the subproblems, SP_n. Once these subproblems have been solved, columns are constructed and added to the master problem.

JuDGE provides several benefits to the user. First, rather than composing the full problem in JuMP, the user is only required to write the subproblems, and describe the structure of the scenario tree. This greatly reduces the possibility of errors when describing the problem, and avoids having to implement non-anticipativity constraints. Moreover, large-scale instances of P are generally impossible to solve as a single optimization problem, and some form of decomposition is needed. JuDGE enables this by relieving the user of the burden of implementing a bespoke decomposition method for their particular problem.

3.1.1. Scenario trees in JuDGE A cumbersome part of modeling and constructing an optimization problem is the input of data into the model. This is especially the case in multistage and stochastic optimization problems. To help define the decision process and organise the relevant data, JuDGE provides a native structure called a `JuDGE.Tree`.

The `JuDGE.Tree` is an essential part of JuDGE; it allows us to describe the structure of the multistage decision problem being optimized. It is important to note here that in our problem formulation, we make no assumptions on the structure of the stochastic process in our problem (i.e. stagewise independence or Markov), which means that we must construct the `JuDGE.Tree` explicitly, node by node. However, several functions are provided to automate the construction of trees:

(a) balanced trees where every non-leaf node has the same degree can be created with a single function call;

(b) trees can be defined by the user and input using a CSV file.

For (a) JuDGE provides the function `narytree(depth::Int64, degree::Int64)`, which creates a symmetric tree. Note that setting the depth to 0 will produce tree with a single node, regardless of the degree.

For (b) JuDGE provides the function `tree_from_file(filename::String)`. This function reads a CSV file that imports the structure of the tree and the corresponding data for each node. For example, nodes 1-9 of the tree in Figure 1 file could be formatted as follows (the white-space is for presentation only).

```
n, p, depth, degree
1, -, 0,     3
2, 1, 1,     2
3, 1, 1,     1
4, 1, 1,     2
5, 2, 2,     2
6, 2, 2,     1
7, 3, 2,     2
8, 4, 2,     1
9, 4, 2,     3
```

The columns `n` and `p` are required, and specify the name of the node, and the predecessor (– for the root node), respectively. Additional columns are optional / arbitrary and JuDGE automatically constructs dictionaries enabling the data to be referenced by node. For example the `depth` of node n can be accessed as follows:

```
data[:depth][n]
```

Within JuDGE, a reference to the root node of a `JuDGE.Tree` gives access to the entire tree, so we need not directly store arrays or dictionaries of nodes. To avoid creating restrictive additional data structures or fields to store probability or subproblem data, we enable

users to create their own dictionary data structures, indexed by the nodes of the tree. This ensures users have full control over the organization of the data that is used in the subproblems, while at the same time, abstracting away the complexities of managing the scenario tree.

When using some of the more advanced features of JuDGE it may be necessary to reference parents or children of nodes, or other specific nodes in the tree. Each node in the `JuDGE.Tree` has a field `children`, which is an array of nodes, which are each either the root of their own subtree, or a `Leaf` node. Individual nodes can be retrieved using the `get_node` function, which takes as arguments, the root node of the tree, and an array of integers, defining the index of the child nodes at each branch. For example, [1,2,1,2] would denote node 14 in Figure 1. JuDGE also provides a number of functions that, given the root node of a tree, return functions that are mainly used within JuDGE itself, but can also be utilized by users.

- `JuDGE.depth`: Given the root node of the tree, this function can be used to find the depths, $\delta(n)$, of any nodes n in the tree.
- `JuDGE.history`: Given the root node of the tree, this function enables the user to find the set of predecessors, $\mathcal{P}_n$, of any node n in the tree.

3.1.2. Constructing JuDGE subproblems Once a scenario tree has been set up in JuDGE and data structures have been established, using dictionaries to map nodes to the appropriate data, we must define a subproblem at each node as a JuMP model. In what follows we will call this problem `sp`.

In order for JuDGE to process these JuMP models and automatically construct the master problem, JuDGE defines additional macros that enable the user to specify expansion variables and corresponding costs. These expansion variables z_n for SP_n (and the corresponding variables x_n for RMP) are defined using the macro:

```
@expansion(sp, scalar_variable)
```

This macro can define sets of variables, in the same way that the `@variable(...)` macro works in JuMP. Thus we could write

```
@expansion(sp, vector_variable[1:n])
```

Once the expansion variables are defined, we specify the costs of choosing these expansions at each node n. These costs is declared in the definition of the subproblem sp for convenience, but JuDGE uses these costs to set the objective of the expansions in the master problem:

```
@capitalcosts(sp, scalar_variable*scalar_cost[node]
    +sum(vector_variable[i]*vector_costs[i] for i in 1:n)
```

Some expansions may create ongoing (e.g. maintenance) costs that can be declared in a similar way:

```
@ongoingcosts(sp, scalar_variable*scalar_cost[node]
    +sum(vector_variable[i]*vector_costs2[i] for i in 1:n)
```

Together, the capital costs and the ongoing costs are used to define c_n in RMP.

Note that the master problem RMP will not force columns returned by subproblems to utilise an investment that has been made an an earlier node. To ensure subproblem formulations are consistent with this assumption, JuDGE expansion variables must appear in `sp` on the right-hand side (with a positive coefficient) of a $\leq$ constraint. This is automatically checked by JuDGE when the `JuDGEModel` is defined, and an error will be thrown if the subproblem is incorrectly specified. Other checks, such as that the same set of expansions are available at every node are also carried out.

Finally, the objective functions of the subproblems are set using the JuMP macro:

```
@objective(model::JuMP.Model, Min, expr::AffExpr)
```

3.1.3. Defining and solving a `JuDGEModel` The default implementation of JuDGE requires four inputs to define a `JuDGEModel` object. These are an instance of `JuDGE.Tree`, a dictionary of probabilities for each node in the tree, a function which constructs a subproblem for each node in the tree, and a reference to a solver. These four components are used to construct the master problem and establish the data structures necessary to perform Dantzig-Wolfe decomposition on the problem. For models with a constant discount factor for every node in the tree, an optional argument `discount_factor` can be set to some positive value less than or equal to 1.

3.2. Solving a `JuDGEModel`

Once a valid `JuDGEModel` has been constructed, it can be solved. JuDGE's core algorithm is based on a Dantzig-Wolfe decomposition, while exploiting special structure that arises

from the capacity expansion setting. A single iteration of the JuDGE algorithm is as follows:

1. the master problem is solved, giving a primal and dual solution;
2. for every node in the tree:
   (a) update the costs of the expansion variables in the subproblem;
   (b) solve the nodal subproblem with the updated costs;
   (c) construct a new column using the primal solution to the subproblem;
   (d) add the new column to the master problem;
   (e) update lower bound.

The JuDGE solver can be called by simply running the command `JuDGE.solve(model)`. This has a number of optional arguments, enabling the user to customize the solve to terminate as detailed below.

- `rlx_abstol`: The solve terminates if the difference between the current objective of the master problem and the lower bound is less than `rlx_abstol`.
- `abstol`: The solve terminates if the difference between the current objective of the best integer solution found so far by the master problem and the lower bound is less than `abstol`.
- `rlx_reltol`: The solve terminates if the relative difference between the current objective of the best integer solution found so far by the master problem and the lower bound is less than `rlx_reltol`.
- `reltol`: The solve terminates if the relative difference between the current objective of the best integer solution found so far by the master problem and the lower bound is less than `reltol`.
- `duration`: This sets the maximum duration in seconds for the algorithm.
- `iter`: This sets the maximum number of iterations for the algorithm.

Once the model has solved to the desired convergence tolerance, if the model has returned an integer-feasible solution, the user can print the expansions using: `JuDGE.print_expansions(model)`. By default, this will only print the non-zero expansions.

In order to recover the optimal decision variables (i.e. y_n) from each subproblem, its investment decisions must be fixed to the optimal values from the master problem, and the subproblem re-solved. JuDGE automatics this process, with the function

`JuDGE.resolve_subproblems(model)`. The entire solution can then be output to a file as follows: `JuDGE.write_solution_to_file(model,joinpath(@__DIR__,"solution.csv"))`

The JuDGE package provides the following benefits for the user:

- simple construction of scenario trees, using dictionaries to reference nodal data;
- leveraging JuMP syntax to write nodal subproblems;
- automatically constructing the master and subproblems;
- automatically providing the dual solution of the master problem as the expansion costs in the subproblems;
- automatically generating new columns for the master problem, based on the primal solutions of the subproblems;
- automatically formulating a deterministic equivalent JuMP model for comparison and evaluation of JuDGE's relative performance.

4. Illustrative example: a stochastic knapsack problem

In this section, we present a small example showing how to build and solve a multistage optimization problem using JuDGE.

4.1. Problem formulation

We consider a person optimizing a sequence of random knapsack problems. At each stage in the sequence, she must decide which items to collect in her knapsack, as well as whether or not she pays a one-off cost now to expand her knapsack from a collection of expansion options. After she has solved her current knapsack problem, she collects the reward of her knapsack, pays the expansion costs (if any), and her knapsack is then emptied. She then observes the next knapsack problem. Her goal is to minimise the expected cost of knapsack expansions minus the expected rewards over the sequence of knapsack problems. The constraint linking the problem temporally is the fact that once she has expanded her knapsack, she can use the expanded knapsack at any time in the future.

To formulate this problem, consider a scenario tree with nodes $\mathcal{N}$. We have the following problem data:

- $\mathcal{I}$, the set of items in each knapsack problem;
- V_n^i, the volume of item i at node n;
- R_n^i, the reward of item i at node n;
- K, the initial capacity of the knapsack;

- $\mathcal{E}$, the set of knapsack expansions;
- H^e, the volume of expansion e;
- C_n^e, the cost of performing expansion e at node n.

For our problem, we will assume for simplicity that each knapsack problem has the same index set $\mathcal{I}$. Then we have the following decision variables:

- y_n^i, binary decision variable for selecting item i at node n;
- x_n^e, binary decision variable for making expansion e at node n.

We have the following constraints:

- The knapsack constraint:

$$\sum_{i\in\mathcal{I}} V_n^i \times y_n^i \leq K + \sum_{e\in\mathcal{E}} H^e \times x_n^e,\ \forall n \in \mathcal{N}. \tag{3}$$

Further, we have the constraints that are required by JuDGE. First, the expansion variables are binary. Second, each expansion can only be made once in the history of the process. In the context of our problem, this means that

$$x_n^e \in \{0,1\},\ \text{and}\ \sum_{h\in\mathcal{P}_n} x_h^e \leq 1,\ \forall n \in \mathcal{L}, \forall e \in \mathcal{E}. \tag{4}$$

Finally, we have the following objective function

$$\min_{y,x} \sum_{n\in N} \phi_n \left[\sum_{e\in\mathcal{E}} C_n^e \times x_n^e - \sum_{i\in\mathcal{I}} R_n^i \times y_n^i \right]. \tag{5}$$

4.2. Using JuDGE

We will now show how JuDGE can be used to solve the knapsack optimization problem. The first step is to install JuDGE via the following function call in the Julia REPL:

```
julia> ] add "https://github.com/reganbaucke/JuDGE.jl"
```

Once JuDGE is installed, we will build our `JuDGEModel` by creating a new Julia script (called "knapsack.jl"). The complete script is listed in Appendix A. This can be run by

```
include("knapsack.jl")
```

The first statements in the script tell Julia that we want the following libraries in our namespace:

```
using JuDGE, JuMP, Gurobi

env = Gurobi.Env()
```

JuDGE provides many different ways for the user to define their tree, but for this example we will use the function `narytree` to construct a tree with depth 2 and degree 2, i.e. `mytree = narytree(2,2)` This creates a 7-node tree, with 4 leaf nodes.

JuDGE does not prescribe how the data should be managed, however, it needs to be able to be referenced based on the node of the tree. This can be achieved through the use of functions which take the node of the tree as one of its arguments, or dictionaries that are indexed by the nodes of the tree.

In our version of the optimization problem, we have 5 items to choose to fill our knapsack with, and we have 6 different expansion options i.e. $|\mathcal{I}| = 5$ and $|\mathcal{E}| = 6$.

We directly specify the investment costs, item volumes and item rewards as dictionaries, with the elements ordered based on a breath-first search of the tree.

```
invest_cost = Dict(zip(collect(mytree,order=:breadth), [15, 8, 8, 4, 4, 4, 4] ))

item_volume = Dict(zip(collect(mytree,order=:breadth), [ [4, 3, 3, 1, 2],
                                                         [5, 3, 4, 2, 1],
                                                         [5, 4, 2, 7, 2],
                                                         [5, 4, 1, 8, 2],
                                                         [3, 1, 5, 6, 3],
                                                         [2, 5, 8, 4, 6],
                                                         [7, 5, 4, 2, 3] ] ))

item_reward = Dict(zip(collect(mytree,order=:breadth), [ [32, 9, 9, 4, 8],
                                                         [30, 12, 40, 10, 9],
                                                         [20, 28, 12, 42, 12],
                                                         [40, 28, 9, 24, 10],
                                                         [15, 7, 20, 48, 12],
                                                         [10, 30, 54, 32, 30],
                                                         [32, 25, 24, 14, 24] ] ))
```

We will also define some data that are constant across all nodes:

```
num_items=5
num_invest=6
initial_volume = 6
invest_volume = [2,2,2,3,3,3]
```

Here `num_items` specifies the number of items available for selection in each subproblem; `num_invest` is the number of investment options that JuDGE can select; `initial_volume` is the volume of the knapsack prior to any investments; and `invest_volume` is the amount that each of the six investments increases the size of the knapsack.

These data are sufficient for us to define our subproblems as JuMP models using some JuDGE-specific macros.

```
function sub_problems(node)
    sp = Model(optimizer_with_attributes(() -> Gurobi.Optimizer(env),
       "OutputFlag" => 0, "MIPGap" => 0.0))
    @expansion(sp, invest[1:num_invest], Bin)
    @capitalcosts(sp, sum(invest[i]*invest_volume[i]*invest_cost[node]
        for i=1:num_invest))
    @variable(sp, y[1:num_items], Bin)
    @constraint(sp, BagExtension, sum(y[i]*item_volume[node][i]
        for i in 1:num_items) <= initial_volume + sum(invest_volume[i] *
           invest[i]
        for i in 1:num_invest))
    @objective(sp, Min sum(-item_reward[node][i] * y[i] for i in 1:num_items))
    return sp
end
```

Note that the `@expansion` macro works in the same way as the `JuMP.@variable` macro. However `@expansion` will automatically declare the expansion variables to be binary, and declare equivalent variables in the master problem. `@capitalcosts` is provided as a `JuMP.AffExpr` in order to represent the expansion costs to be used in the master problem. JuDGE has been tested and shown to work with several solvers: Gurobi (Gurobi Optimization 2016), CPLEX (IBM 2018), Clp/CBC (CoinOR 2012) and GLPK (Makhorin 2000). As discussed in Singh et al. (2009), using an interior point method (without crossover) to solve RMP generally leads to faster and more consistent convergence of the algorithm. Appropriate settings for each solver are included with the example problems accompanying the JuDGE package.

If we do not have a custom probability distribution for the tree, JuDGE provides a function which sets uniform conditional probabilities throughout the tree. Using this probability function we can define our `JuDGEModel` as follows:

```
model = JuDGEModel(mytree, ConditionallyUniformProbabilities, sub_problems,
    optimizer_with_attributes(() -> Gurobi.Optimizer(env), "OutputFlag" => 0,
    "Method" => 2, "Crossover" => 0,  "MIPGap" => 0.0))
```

We can then solve our model using

```
JuDGE.solve(model)
```

```
Establishing JuDGE model for tree: Subtree rooted at node 1 containing 7 nodes
Checking sub-problem format... Passed
Building master problem... Complete
 Obj Value  |  UpperBound LowerBound |   Abs Diff    Rel Diff |    Fraction |  Time  Iter
        Inf |         Inf       -Inf |        Inf         NaN |         NaN |  0.017     1
  2.250e+02 |   2.250e+02 -2.620e+02 | 4.870e+02  1.859e+00 |  0.000e+00 |  0.031     2
 -1.420e+02 |  -1.420e+02 -2.620e+02 | 1.200e+02  4.580e-01 |  0.000e+00 |  0.046     3
 -1.540e+02 |  -1.540e+02 -2.090e+02 | 5.500e+01  2.632e-01 |  0.000e+00 |  0.061     4
```

```
−1.590e+02 | −1.590e+02 −1.960e+02 | 3.700e+01  1.888e−01 |  0.000e+00 |  0.078    5
−1.625e+02 | −1.590e+02 −1.805e+02 | 1.800e+01  9.972e−02 |  5.000e−01 |  0.094    6
−1.640e+02 | −1.640e+02 −1.690e+02 | 5.000e+00  2.959e−02 |  0.000e+00 |  0.109    7
−1.640e+02 | −1.640e+02 −1.650e+02 | 1.000e+00  6.061e−03 |  0.000e+00 |  0.118    8
−1.640e+02 | −1.640e+02 −1.640e+02 | 0.000e+00  0.000e+00 |  0.000e+00 |  0.127    9
```

The output above shows (from left to right): the current master problem objective value; the objective of the best integer-feasible solution found so far; the lower bound; the absolute and relative difference between the relaxed master problem objective and the lower bound; and maximum fractionality (absolute distance from 0 or 1) of any master problem variable; the current time; and the iteration number. Note that in the 6^{th} iteration, the master problem returns a fractional solution, because of this, the upper bound is not updated in this iteration.

Once the model is solved, we can print the expansions

```
JuDGE.print_expansions(model)
```

```
JuDGE Expansions
Node 11: "invest[4]" 1.0
Node 11: "invest[6]" 1.0
Node 112: "invest[5]" 1.0
Node 12: "invest[4]" 1.0
Node 12: "invest[5]" 1.0
Node 12: "invest[6]" 1.0
Node 121: "invest[1]" 1.0
Node 121: "invest[3]" 1.0
Node 122: "invest[1]" 1.0
Node 122: "invest[2]" 1.0
Node 122: "invest[3]" 1.0
```

By default this will print the non-zero expansions in the master problem, but there is an optional argument `onlynonzero` which can be set to `false` if you want to print all expansion variable values.

This output can be difficult to interpret, since different investments provide different volume increments for the knapsack. To aid the interpretability of JuDGE output, we allow an optional argument `format` which accepts a function that processes the solution before it is printed. Two examples of this functionality are presented in Appendix B. In order to output the full solution, we must first re-solve the subproblems, given the optimal investments, and then we can write the full solution to a file, as shown below.

```
JuDGE.resolve_subproblems(model)

JuDGE.write_solution_to_file(model,joinpath(@__DIR__,"solution.csv"))
```

5. Advanced features of JuDGE

In this section we will discuss several advanced features of JuDGE which enable more complex capacity expansion problems to be modelled, and solved to near global optimality. These features can all be applied together, but for simplicity, we discuss the changes that are made to the master problem individually.

5.1. Branch and price

In many instances of the problem P, the relaxed master problem of the Dantzig-Wolfe decomposition will either return naturally a integer solution, or is able to find an integer solution that is deemed to be sufficiently close to the lower bound that no branching is required. However, for instances where a smaller bound gap is needed, JuDGE provides a branch-and-price method that branches by cloning the master problem and adding additional constraints, then JuDGE performs additional column generation at the cloned nodes. This process is repeated until the best integer-feasible solution is close enough to the smallest lower bound of all cloned master problems.

To use the branch-and-price algorithm you run: `best=branch_and_price(model)` which returns `best`, which is a `JuDGEModel`. You can then print and write the solution to a file as discussed in section 3.1.3. The function `branch_and_price` has an optional argument `search` to customise how the branch and price tree is explored; this can be set to:

- `:depth_first_dive`, which performs a depth-first search;
- `:breadth_first`, which performs a breadth-first search;
- `:lowestLB`, which branches on the node in the tree with the lowest lower bound.

The `branch_and_price` method can take the same optional arguments as outlined in Section 3.2, in order to set termination criteria.

5.2. Lag and duration of expansions

In stochastic capacity expansion problems it is important to accurately model the timing of decisions relative to the revelation of information. In the formulation of P there is an implicit assumption that a capacity expansion decision is made *after* the stage uncertainty is revealed (in a nested wait-and-see manner). This may not be an appropriate model if there is a lag between when an expansion decision is made and when the capacity becomes available.

The other implicit assumption in P is that once an expansion decision is made, it will be available at all subsequent nodes in the tree. This will not be valid for investments

in capacity that have a limited lifetime; we will refer to the number of periods that the expansion is available as its duration. In order to enable JuDGE to model these features, we define a vector Δ_{hn} for $h, n \in \mathcal{N}$ to be a binary vector with rows corresponding to different expansion decisions, defined as follows:

$$e_i^\top \Delta_{hn} = \begin{cases} 1, & \text{if expansion } i \text{ at node } h \text{ grants the corresponding capacity at node } n, \\ 0, & \text{otherwise.} \end{cases}$$

To incorporate this into the master problem, we perform an elementwise multiplication $\otimes$ in constraint (1) in RMP, modifying it as follows:

$$\sum_{j \in \mathcal{J}_n} \hat{z}_n^j w_n^j \leq \sum_{h \in \mathcal{P}(n)} \Delta_{hn} \otimes z'_h, \quad n \in \mathcal{N}.$$

As an example, suppose we are modeling a problem using the tree shown in Figure 1, and have two possible expansions: A and B. Expansion A has a lag of 1, and has an unlimited duration, and expansion B as a lag of 0, but a duration of 2. We can only use expansion A at node 16 if the expansion decision was made at nodes 1, 4 or 9; and can only use expansion B at node 16 if the expansion decision was made at nodes 9 or 16.

Within JuDGE this can be implemented with additional arguments for the `@expansion(...)` macro.

```
@expansion(model, A, Bin, lag=1) #set lag to 1, duration is infinite
@expansion(model, B, Bin, duration=2) #set duration to 2, lag is 0
```

5.3. Side constraints

The standard JuDGE master problem has a very simple structure, with only two non-trivial constraints. Constraint (1) ensures that for an integer-feasible solution, columns can only be selected if the expansion used for that column has been granted by the master (expanded at the current node or some preceding node in the scenario tree), whereas constraint (2) ensures that you must choose a convex combination of columns for each of the nodes.

In order to create more complex models, JuDGE permits the user to easily add additional constraints to the master. These could be used, for example, to:

- limit the number of expansions, or the amount of money spent on expansions in a particular node;

- prevent certain expansions being made at specific nodes in the tree;
- create logical constraints between different types of expansions across the scenario tree; e.g. if one expansion is made, then other expansions become available at subsequent nodes, or alternatively expansions could be mutually exclusive.

This functionality is implemented through the optional argument `sideconstraints` for the JuDGEModel constructor. This argument accepts a function, taking arguments `model` (which will be a reference to the master problem), and `tree` (which is the entire scenario tree). An example of this could be that we wish to impose a budget constraint in our stochastic knapsack problem, so we can only spend 40 per node. (This type of constraint would be invalid within the subproblem.) This is implemented using the function

```
function budget(model,tree)
  for n in collect(tree)
     @constraint(model,invest_cost[n]*sum(invest_volume[i]*invest[n][i] for i
        in 1:num_invest)<=40)
  end
end
```

Note that we reference the variables defined in the `sub_problems(node)` function, with the addition of a reference to the node of the subproblem. In order words the variable `invest[i]` defined in the subproblem for node `n` is referenced by `invest[n][i]`. With these side constraints defined, we create a new `JuDGEModel`, setting the optional argument `sideconstraints` to `budget`.

```
model = JuDGEModel(mytree, ConditionallyUniformProbabilities, sub_problems,
    optimizer_with_attributes(() -> Gurobi.Optimizer(env), "OutputFlag" => 0,
    "Method" => 2, "Crossover" => 0,  "MIPGap" => 0.0),
    sideconstraints=budget)
```

5.4. Shutdown decisions

Constraint (1) in RMP is a set-covering constraint which ensures that a subproblem at node n can only use an expansion that is made at some predecessor node (or the current node) $h \in \mathcal{P}(n)$. Due to this being an inequality constraint, it is possible that this will not bind at the optimal solution. What this means from a practical perspective is that a subproblem can attain a better objective value by not making use of an expansion that is selected by the master problem. There are a few circumstances where this could occur, but the simplest is when we want to model a shutdown decision, rather than an expansion. This is not able to be modeled using RMP, since any shutdown action would be ignored by the subproblem.

Shutdowns therefore must be modeled differently in the master. The way that this has been implemented in JuDGE involves a modification of constraint (1), with the inclusion of a diagonal matrix Θ, defined as follows:

$$\Theta_{ii} = \begin{cases} 1 & \text{if the } i^{\text{th}} \text{ element of } x_n \text{ corresponds to an expansion decision,} \\ -1 & \text{if the } i^{\text{th}} \text{ element of } x_n \text{ corresponds to a shutdown decision.} \end{cases}$$

$$\begin{aligned} \text{RMP2:} \qquad \min \quad & \sum_{n\in\mathcal{N}} \phi_n c_n^\top x_n + \sum_{n\in\mathcal{N}} \sum_{j\in J_n} \phi_n \psi_n^j w_n^j \\ \text{s.t.} \quad & \Theta \sum_{j\in\mathcal{J}_n} \hat{z}_n^j w_n^j \le \Theta \sum_{h\in\mathcal{P}(n)} x_h, \quad n\in\mathcal{N}, && [\pi_n] \\ & \sum_{j\in\mathcal{J}_n} w_n^j = 1, \quad n\in\mathcal{N}, && [\mu_n] \\ & w_n^j \ge 0, \quad n\in\mathcal{N}, j\in\mathcal{J}_n, \\ & x_n \ge 0, \quad n\in\mathcal{N}. \end{aligned}$$

With this change, we see that expansions will be constrained as normal, but if the i^{th} element of x_n corresponds to a shutdown the constraint for i effectively becomes:

$$e_i^\top \sum_{j\in\mathcal{J}_n} \hat{z}_n^j w_n^j \ge e_i^\top \sum_{h\in\mathcal{P}(n)} x_h, \quad n\in\mathcal{N}. \tag{6}$$

For integer solutions to the master problem, this means that prior to the shutdown occurring, columns with a 0 in the corresponding position (meaning that the subproblem has not activated the shutdown) or a 1 (meaning that the shutdown has occurred) are permitted. However, after the shutdown, only columns with a 1 will be permitted.

If setting a shutdown variable to 1 leads to a lower availability of resources in the subproblem, then the constraint (6) in the master will typically bind at the optional solution, since a subproblem would only make that resource unavailable if it is shutdown in the master.

To implement shutdown decisions in JuDGE you must define the variables in the subproblem, using the `@shutdown(...)` macro, which is analogous to the `@expansion(...)` macro.

The master problem's costs associated with a shutdown at node n, c_n are defined in the same way as the expansions, using the `@capitalcosts(...)` and `@ongoingcosts(...)` macros. However, of course, for a shutdown, we would be modelling the net salvage value and avoided maintenance costs, so these costs will typically be negative.

5.5. Conditional Value at Risk

JuDGE implements risk-aversion through a CVaR objective applied to the overall tree. The scenarios of the tree correspond to the leaf nodes, so for each leaf node $l \in \mathcal{L}$ we evaluate the overall cost over the predecessor nodes $\mathcal{P}_l$, we denote this ξ_l. Specifically, we seek to minimize a convex combination of expectation and CVaR:

$$(1-\lambda)\mathbb{E}[\xi] + \lambda \text{CVaR}_{1-\alpha}(\xi).$$

Using the standard linear programming formulation of CVaR (Rockafellar and Uryasev 2002), for $\text{CVaR}_{1-\alpha}(\xi)$ we define the $(1-\alpha)$-quantile to be η, and reformulate the master problem as follows:

$$\begin{aligned}
\text{RMP-CVaR:} \quad \min \quad & \sum_{l\in L} \phi_l \xi_l + \frac{\lambda}{\alpha} \sum_{l\in\mathcal{L}} \phi_l \left((1-\alpha)v_l + \alpha v'_l\right) \\
\text{s.t.} \quad & \xi_l = \sum_{n\in\mathcal{P}_l} c_n^\top x_n + \sum_{n\in\mathcal{P}_l} \sum_{j\in J_n} \psi_n^j w_n^j, \quad l \in \mathcal{L}, \\
& \xi_l = \eta + v_l - v'_l, \quad l \in \mathcal{L}, \\
& \sum_{j\in\mathcal{J}_n} \hat{z}_n^j w_n^j \le \sum_{h\in\mathcal{P}(n)} x_h, \quad n \in \mathcal{N}, && [\pi_n] \\
& \sum_{j\in\mathcal{J}_n} w_n^j = 1, \quad n \in \mathcal{N}, && [\mu_n] \\
& v_l, v'_l \ge 0, \quad l \in \mathcal{L}, \\
& w_n^j \ge 0, \quad n \in \mathcal{N}, j \in \mathcal{J}_n, \\
& x_n \ge 0, \quad n \in \mathcal{N}.
\end{aligned}$$

The choice $\lambda = 0$ or $\alpha = 0$ corresponds to a pure expectation objective, whereas $\lambda = 1$ will correspond to an objective that only minimizes $\text{CVaR}_{(1-\alpha)}(\xi)$. Note that since risk aversion is applied over the scenarios, this amounts to using a nested risk measure, while changing values of α throughout the tree, depending on the accumulated cost up to each node. This is different from optimizing a classical nested risk measure, for which the risk attitude defined by λ and α at each node does not depend on previous outcomes.

To specify a conditional-value-at-risk objective function we provide the optional argument `CVaR` to the JuDGEModel constructor; this argument should be provided as an ordered pair, with the first element specifying the weight λ on CVaR, and the second the CVaR quantile α. For example, the code below will set $\lambda = 0.5$ and $\alpha = 0.1$.

```
model = JuDGEModel(..., CVaR = (0.5, 0.9))
```

6. Benchmarking JuDGE

In order to demonstrate the computational benefits of solving multistage optimization problems using JuDGE, we have performed a number of tests comparing the performance of open-source and commercial solvers, first as part of the Dantzig-Wolfe decomposition, and second applied to deterministic equivalent formulations.

Our benchmarks will be performed on a version of the knapsack problem presented in section 4.2, with 200 items, 6 possible expansions, and scenario trees of ranging from 7 nodes to 3906 nodes. The items have randomized rewards and volumes, which are reproduced using 5 random seeds. Each instance of the problem was solved with a 2-hour time limit and in Table 6, below, we report the solve time to both 1% and 0.1% optimality or the optimality gap after 2 hours. All benchmarks were run on a VM with 16 virtual processors of a CPU Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz with 128GB of RAM, running Julia 1.3.1, Gurobi 9.02, GLPK 4.64, JuMP 0.21.3 and JuDGE 0.60.

For the models with a small tree (depth 2), all the models were able to be solved in under 10 seconds to a 0.1% optimality gap. However, for the models with a tree of depth and degree equal to 3, we find that for some instances the deterministic equivalent takes 1-4 minutes to get to 1% and 10-20 minutes to achieve a 0.1% optimality gap, in one case it was unable to solve to this tolerance within 2 hours. However, the models solved using JuDGE's Dantzig-Wolfe implementation are all solved to 0.1% optimality in under 40 seconds. For all instances with larger trees, the deterministic equivalent could not be solved to 0.1% optimality, and in two instances failed to solve to a 1% bound gap within the 2-hour time-limit.

For most problem sizes JuDGE performed similarly with the two solvers. However, for the very large instances (depth 5), we see that those instances solved using Gurobi failed to reach the 0.1% MIP gap within 2 hours, whereas all instances solved using with GLPK were solved, with 4 of the 5 solved in under 1 hour. Moreover, those instances solved using GLPK took less than 15 minutes to solve to 1% optimality, with the same problems taking over an hour when using Gurobi. We observe that this performance difference is due to GLPK outperforming Gurobi on the MIP subproblems for this knapsack problem.

These results show that JuDGE enables open-source solvers to outperform commerical solvers for certain classes of problem. Moreover, due to JuDGE's flexibility, building on top of JuMP, large-scale capacity planning models can now be developed without the need

					0.1%			1.0%		
					JuDGE		DetEq	JuDGE		DetEq
Degree	Depth	Seed	Nodes	Variables	Gurobi	GLPK	Gurobi	Gurobi	GLPK	Gurobi
2	2	1	7	1495	1.2s	1.0s	1.2s	1.2s	0.9s	0.2s
2	2	2	7	1495	0.9s	1.1s	0.9s	0.9s	1.1s	0.2s
2	2	3	7	1495	2.1s	1.4s	2.5s	2.1s	1.4s	0.2s
2	2	4	7	1495	0.9s	1.1s	1.8s	0.9s	1.1s	0.2s
2	2	5	7	1495	1.5s	1.4s	3.8s	1.3s	0.9s	0.8s
3	2	1	15	2778	4.8s	3.9s	5.4s	0.3s	2.1s	0.4s
3	2	2	15	2778	4.7s	4.0s	5.0s	3.5s	2.8s	0.4s
3	2	3	15	2778	2.2s	2.2s	5.1s	1.8s	1.9s	0.8s
3	2	4	15	2778	3.9s	3.8s	15.5s	2.8s	2.8s	1.7s
3	2	5	15	2778	2.1s	2.0s	3.3s	1.3s	1.6s	0.3s
3	3	1	40	8547	22s	28.90s	485.72s	10s	15s	80s
3	3	2	40	8547	28s	36.15s	1049.62s	17s	29s	122s
3	3	3	40	8547	37s	26.86s	53.20s	26s	23s	35s
3	3	4	40	8547	20s	23.77s	11.77s	7s	9s	2s
3	3	5	40	8547	23s	24.06s	0.36%	13s	17s	223s
3	4	1	85	18169	49s	41.44s	0.51%	21s	38s	116s
3	4	2	85	18169	82s	83.77s	0.74%	38s	45s	1855s
3	4	3	85	18169	102s	91.32s	1.75%	48s	42s	1.75%
3	4	4	85	18169	112s	89.37s	0.58%	40s	69s	845s
3	4	5	85	18169	54s	136.52s	0.29%	18s	22s	799s
4	4	1	341	72889	382s	565.38s	0.35%	86s	100s	758s
4	4	2	341	72889	250s	343.34s	0.32%	85s	94s	877s
4	4	3	341	72889	613s	656.49s	0.30%	116s	165s	572s
4	4	4	341	72889	327s	288.48s	0.34%	82s	104s	665s
4	4	5	341	72889	617s	459.89s	0.23%	83s	132s	577s
4	5	1	781	166978	1710s	1637.48s	0.96%	811s	1059s	5589s
4	5	2	781	166978	1906s	2280.90s	0.73%	337s	1165s	905s
4	5	3	781	166978	1429s	1614.45s	0.51%	353s	990s	5265s
4	5	4	781	166978	1756s	1637.31s	0.55%	559s	471s	3335s
4	5	5	781	166978	376s	273.93s	0.18%	376s	274s	641s
5	5	1	3906	835103	0.11%	2247.03s	0.94%	6358s	639s	2840s
5	5	2	3906	835103	0.15%	5407.42s	0.89%	3438s	753s	3080s
5	5	3	3906	835103	0.26%	3139.88s	0.90%	4379s	738s	4848s
5	5	4	3906	835103	6740s	2198.67s	0.39%	3616s	745s	3152s
5	5	5	3906	835103	0.16%	3049.85s	1.07%	3688s	753s	1.07%

to derive the necessary non-anticipativity constraints to model the uncertainty, nor create a bespoke decomposition implementation.

Acknowledgments

We are grateful to Sonja Wolgrin and Ryan Cory-Wright for useful discussions in the development and computational testing of JuDGE.

Appendix A: Script for `knapsack.jl`

```
using JuDGE, JuMP, Gurobi

env = Gurobi.Env()

# parameters for each node in the tree
invest_cost = Dict(zip(collect(mytree,order=:breadth), [15, 8, 8, 4, 4, 4, 4] ))


item_volume = Dict(zip(collect(mytree,order=:breadth), [ [4, 3, 3, 1, 2],
                                                         [5, 3, 4, 2, 1],
                                                         [5, 4, 2, 7, 2],
                                                         [5, 4, 1, 8, 2],
                                                         [3, 1, 5, 6, 3],
                                                         [2, 5, 8, 4, 6],
                                                         [7, 5, 4, 2, 3] ] ))

item_reward = Dict(zip(collect(mytree,order=:breadth), [ [32, 9, 9, 4, 8],
                                                         [30, 12, 40, 10, 9],
                                                         [20, 28, 12, 42, 12],
                                                         [40, 28, 9, 24, 10],
                                                         [15, 7, 20, 48, 12],
                                                         [10, 30, 54, 32, 30],
                                                         [32, 25, 24, 14, 24] ] ))

# parameters shared by every node in the tree
num_items=5
num_invest=6
initial_volume = 6
invest_volume = [2,2,2,3,3,3]

# function that defines the problem at each node
function sub_problems(node)
    sp = Model(optimizer_with_attributes(() -> Gurobi.Optimizer(env), "OutputFlag"
        => 0, "MIPGap" => 0.0))
    @expansion(sp, invest[1:num_invest], Bin)
    @capitalcosts(sp, sum(invest[i]*invest_volume[i]*invest_cost[node]
        for i=1:num_invest))
    @variable(sp, y[1:num_items], Bin)
    @constraint(sp, BagExtension, sum(y[i]*item_volume[node][i]
        for i in 1:num_items) <= initial_volume + sum(invest_volume[i] * invest[i]
        for i in 1:num_invest))
    @objective(sp, Min, sum(-item_reward[node][i] * y[i] for i in 1:num_items))
    return sp
end

# create the JuDGE model
model = JuDGEModel(mytree, ConditionallyUniformProbabilities, sub_problems,
    optimizer_with_attributes(() -> Gurobi.Optimizer(env), "OutputFlag" => 0,
    "Method" => 2, "Crossover" => 0,  "MIPGap" => 0.0))

# solve the JuDGE model
JuDGE.solve(model)

# display the investments and write the solution to a file
JuDGE.print_expansions(model)
JuDGE.resolve_subproblems(model)
JuDGE.write_solution_to_file(model,joinpath(@__DIR__,"solution.csv"))
```

Appendix B: Formatting JuDGE output

By default this will print the non-zero expansions in the master problem, but there is an optional argument `onlynonzero` which can be set to `false` if you want to print all expansion variable values.

This output is difficult to interpret, since the various investments provide differing volume increments for the knapsack. To aid the interpretability of this output, we allow an optional argument `format` which accepts a function that processes the solution before it is printed.

```
function format_output(s::Symbol,value)
  output=Dict{Int64,Float64}()
  if s==:invest
     for i in 1:num_invest
        output[i]=invest_volume[i]*value[i]
     end
     return output
  end
  return nothing
end

JuDGE.print_expansions(model, format=format_output)
```

The output is now:

```
JuDGE Expansions
Node 11: "invest[2]" 2.0
Node 11: "invest[3]" 2.0
Node 11: "invest[1]" 2.0
Node 112: "invest[6]" 3.0
Node 12: "invest[4]" 3.0
Node 12: "invest[5]" 3.0
Node 12: "invest[6]" 3.0
Node 121: "invest[3]" 2.0
Node 121: "invest[1]" 2.0
Node 122: "invest[2]" 2.0
Node 122: "invest[3]" 2.0
Node 122: "invest[1]" 2.0
```

This shows the volume expanded by each investment. However, if we want to see how much is being expanded in total, we can use a different function to format the output:

```
function format_output(s::Symbol,value)
  if s==:invest
     return sum(invest_volume[i]*value[i] for i in 1:num_invest)
  end
  return nothing
end

JuDGE.print_expansions(model, format=format_output)
```

With this new format the output becomes:

```
JuDGE Expansions
Node 11: "invest" 6.0
Node 112: "invest" 3.0
Node 12: "invest" 9.0
Node 121: "invest" 4.0
Node 122: "invest" 6.0
```

Now we can directly see how much capacity was added at each node.

References

Ahmed S, King AJ, Parija G (2003) A multi-stage stochastic integer programming approach for capacity expansion under uncertainty. *Journal of Global Optimization* 26(1):3–24.

Borison AB, Morris PA, Oren SS (1984) A state-of-the-world decomposition approach to dynamics and uncertainty in electric utility generation expansion planning. *Operations Research* 32(5):1052–1068.

Carøe C, Schultz R (1999) Dual decomposition in stochastic integer programming. *Operations Research Letters* 24(1-2):37–45.

CoinOR (2012) COmputational INfrastructure for Operations Research. URL `http://www.coin-or.org`.

Dentcheva D, Römisch W (2004) Duality gaps in nonconvex stochastic optimization. *Mathematical Programming* 101(3):515–535.

Dowson O (2017) SDDP in Julia. Technical report, University of Auckland.

Dunning I, Huchette J, Lubin M (2017) JuMP: A modeling language for mathematical optimization. *SIAM Review* 59(2):295–320.

Flores-Quiroz A, Palma-Behnke R, Zakeri G, Moreno R (2016) A column generation approach for solving generation expansion planning problems with high renewable energy penetration. *Electric Power Systems Research* 136:232–241.

Govindan K, Fattahi M, Keyvanshokooh E (2017) Supply chain network design under uncertainty: A comprehensive review and future research directions. *European Journal of Operational Research* 263(1):108–141.

Gurobi Optimization I (2016) Gurobi optimizer reference manual. URL `http://www.gurobi.com`.

Hong X, Lejeune MA, Noyan N (2015) Stochastic network design for disaster preparedness. *IIE Transactions* 47(4):329–357.

IBM (2018) IBM ILOG CPLEX Optimization Studio. URL `https://www.ibm.com/products/ilog-cplex-optimization-studio`.

Kaut M, Midthun KT, Werner AS, Tomasgard A, Hellemo L, Fodstad M (2014) Multi-horizon stochastic programming. *Computational Management Science* 11(1-2):179–193.

Luss H (1982) Operations research and capacity expansion problems: A survey. *Operations Research* 30(5):907–947.

Makhorin A (2000) The GNU linear programming kit (GLPK). GNU Software Foundation. URL `http://www.gnu.org/software/glpk/glpk.html`.

Mete HO, Zabinsky ZB (2010) Stochastic optimization of medical supply location and distribution in disaster management. *International Journal of Production Economics* 126(1):76–84.

Moreno R, Street A, Arroyo JM, Mancarella P (2017) Planning low-carbon electricity systems under uncertainty considering operational flexibility and smart grid technologies. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375(2100):20160305.

Munoz FD, Watson JP (2015) A scalable solution framework for stochastic transmission and generation planning problems. *Computational Management Science* 12(4):491–518.

Özaltın OY, Prokopyev OA, Schaefer AJ (2018) Optimal design of the seasonal influenza vaccine with manufacturing autonomy. *INFORMS Journal on Computing* 30(2):371–387.

Rockafellar RT, Uryasev S (2002) Conditional value-at-risk for general loss distributions. *Journal of Banking & Finance* 26(7):1443–1471.

Rockafellar RT, Wets RJB (1991) Scenarios and policy aggregation in optimization under uncertainty. *Mathematics of operations research* 16(1):119–147.

Sen S, Sherali H (2006) Decomposition with branch-and-cut approaches for two-stage stochastic mixed-integer programming. *Mathematical Programming* 106(2):203–223.

Singh KJ, Philpott AB, Wood RK (2009) Dantzig-Wolfe decomposition for solving multistage stochastic capacity-planning problems. *Operations Research* 57(5):1271–1286.

Snyder L (2006) Facility location under uncertainty: a review. *IIE transactions* 38(7):547–564.

Van Mieghem JA (2003) Commissioned paper: Capacity management, investment, and hedging: Review and recent developments. *Manufacturing & Service Operations Management* 5(4):269–302.

Villumsen JC, Brønmo G, Philpott AB (2012) Line capacity expansion and transmission switching in power systems with large-scale wind power. *IEEE Transactions on Power Systems* 28(2):731–739.

Watson JP, Woodruff DL (2011) Progressive hedging innovations for a class of stochastic mixed-integer resource allocation problems. *Computational Management Science* 8(4):355–370.

Wogrin S, Downward A, Tejada-Arango D, Philpott AB (2020) Transmission capacity expansion using JuDGE. *Electric Power Optimization Centre,* URL `www.epoc.org.nz`.

Wu A, Philpott AB, Zakeri G (2017) Investment and generation optimization in electricity systems with intermittent supply. *Energy Systems* 8(1):127–147.