– Prototype Manual –

`Siplib.jl`: A `Julia` package for SIPLIB 2.0

Yongkyu Cho
IME, POSTECH

June 22, 2018

1 Introduction

SIPLIB 2.0 is an extended version of SIPLIB, the first SIP-oriented instance collection constructed in 2002 by Shabbir Ahmed and his colleagues [1]. We implement a `Julia` package for users to utilize new funtionalities of SIPLIB 2.0.

In this manual, we introduce the package. The main functionality of `Siplib.jl` is to generate SMPS files of instances of stochastic integer programming (SIP) problems. Currently supported problems and corresponding instance names are summarized in Table 1 and Table 2.

Table 1: Problems in SIPLIB 2.0

Problem	Description	Main reference
DCAP	Dynamic capacity planning with stochastic demand	Ahmed and Garcia [2]
MPTSPs	Multi-path traveling salesman problem with stochastic travel costs	Tadei et al. [3]
SIZES	Optimal product substitution with stochastic demand	Jorjani et al. [4]
SMKP	Stochastic multiple knapsack problem	Angulo et al. [5]
SSLP	Stochastic server location problem	Ntaimo and Sen [7]
SUC	Stochastic unit commitment problem	Papavasiliou and Oren [6]

Table 2: Instance naming rules

Problem	Instance name	Remark
DCAP	DCAP_R_N_T_$\mathcal{S}$	R: number of resources, N: number of tasks, T: number of time periods, $\mathcal{S}$: number of scenarios
MPTSPs	MPTSPs_D_N_$\mathcal{S}$	D: node distribution strategy, N: number of nodes, $\mathcal{S}$: number of scenarios
SIZES	SIZES_$\mathcal{S}$	$\mathcal{S}$: number of scenarios
SMKP	SMKP_I_$\mathcal{S}$	I:number of types for item, $\mathcal{S}$: number of scenarios
SSLP	SSLP_I_J_$\mathcal{S}$	I: number of clients, J: number of server locations, $\mathcal{S}$: number of scenarios
SUC	SUC_D_$\mathcal{S}$	D: day type, $\mathcal{S}$: number of scenarios

1.1 Preliminaries

1.1.1 SMPS format

SMPS format is a data conventions for the automatic input of multiperiod stochastic linear programs. The input format is based on an old column-oriented format MPSX standard and is designed to promote the efficient conversion of originally deterministic problems by introducing stochastic variants in separate files.

Three input files are required to specify an stochastic program (SP) in SMPS format:

- `.cor`: Core file written in MPS format. This describes the fundamental problem structure and contains the first-stage data and one second-stage scenario data.
- `.tim`: Time file which specifies the location where the second-stage begins.
- `.sto`: Stoch file which contains stochastic data of all scenarios except the one included in `.cor` file.

One of the main functions of the package is to generate SMPS files for an instance. For example, the package generates the following three files for a DCAP instance DCAP_R_N_T_S.

- `DCAP_R_N_T_S.cor`
- `DCAP_R_N_T_S.tim`
- `DCAP_R_N_T_S.sto`

1.1.2 `JuMP.Model`-type object

`JuMP.Model`-type object is an object that contains every information of an instance. Hence, almost every function in the package requires this object as one of its input arguments. Combining StructJuMP package together with JuMP package, constructing a `JuMP.Model` object is quite simpler and more intuitive than any other algebraic modeling language dedicated for SIP. For example, the script in Fig. 4 constructs an object `model` of a DCAP_R_N_T_S instance.

1.2 Structure of the `Julia` package

The tree in Fig. 1 shows how the `Julia` package is structured.

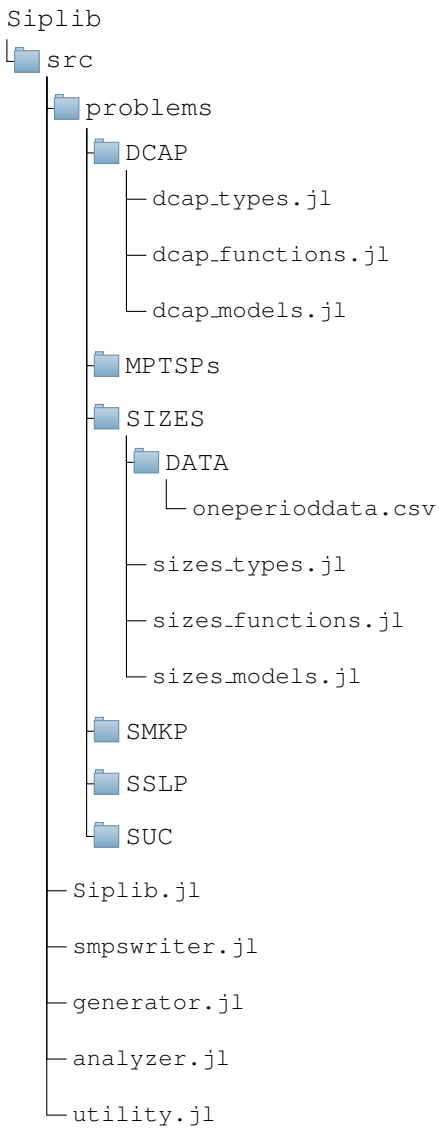


Figure 1: Structure of the `Julia` package

`src` folder on the top contains every implementation of the package. In the directly descendant folder `problems`, another folders with names for each problem present. Some folders in `problems` (e.g., `SIZES`) have `DATA` folder in it which contains external data for generating the instances.

1.2.1 Three core scripts: `types`, `functions`, `models`

Each folder in `problems` contains three scripts (types, functions, models) for constructing `JuMP.Model` object for corresponding problem. Each kind of script has independent role that defines

`types` This kind of script defines the *composite type* (also known as structure or aggregate data type in various languages) that is used to store data for constructing `JuMP.Model` object. The object of this composite type will contain all set and parameter data that define an instance. For example, `dcap_types.jl` in Fig. 2, defines the composite type `DCAPData`.

```
mutable struct DCAPData
    # Sets
    R # set of resources
    N # set of tasks
    T # set of time periods
    S # set of scenarios

    # Parameters
    a # a[i,t]: variable cost for expanding capacity of resource i at time t
    b # b[i,t]: fixed cost for expanding capacity of resource i at time t
    c # c[i,j,t,s]: cost of processing task j using resource i in period t under scenario s
    c0 # c0[j,t,s]: penalty cost of failing to assign a resource to task j under scenario s
    d # d[j,t,s]: processing requirement for task j in period t under scenario s
    Pr # Pr[s]: probability of occurence of scenario s

    DCAPData() = new()
end
```

Figure 2: Example: `dcap_types.jl`

`functions` This kind of script mostly defines the functions exploited for generating random data for each problem. `dcap_functions.jl` in Fig 3, for example, defines a function that returns the `DCAPData`-type object as well as generates random data.

```
function DCAPData(nR::Int, nN::Int, nT::Int, nS::Int, seed::Int=1)::DCAPData

    srand(seed)

    data = DCAPData()

    data.R = 1:nR
    data.N = 1:nN
    data.T = 1:nT
    data.S = 1:nS

    # generate data
    data.a = rand(nR, nT) * 5 + 5
    data.b = rand(nR, nT) * 40 + 10
    data.c = rand(nR, nN, nT, nS) * 5 + 5
    data.c0 = rand(nN, nT, nS) * 500 + 500
    data.d = rand(nN, nT, nS) + 0.5
    data.Pr = ones(nS)/nS

    return data

end
```

Figure 3: Example: `dcap_functions.jl`

`models` This kind of script contains a definition of the function that finally constructs and returns `JuMP.Model`-type object of an instance. This function calls the members in JuMP and StructJuMP packages. The two kinds of scripts above should be included in this script. Script in Fig 4 shows the example of DCAP.

```
include("./dcap_types.jl")
include("./dcap_functions.jl")

function DCAP(nR::Int, nN::Int, nT::Int, nS::Int, seed::Int)::JuMP.Model

    # generate instance data
    data = DCAPData(nR, nN, nT, nS)

    R, N, T, S = data.R, data.N, data.T, data.S
    a, b, c, c0, d, Pr = data.a, data.b, data.c, data.c0, data.d, data.Pr

    # construct the model
    model = StructuredModel(num_scenarios = nS)

    ## 1st stage
    @variable(model, x[i=R,t=T] >= 0)
    @variable(model, u[i=R,t=T], Bin)
    @objective(model, Min,
    sum(a[i,t]*x[i,t] + b[i,t]*u[i,t] for i in R for t in T)
    )
    @constraint(model, [i=R,t=T], x[i,t] - u[i,t] <= 0)

    ## 2nd stage
    for s in S
    sb = StructuredModel(parent=model, id = s, prob = Pr[s])
    @variable(sb, y[i=R, j=N, t=T], Bin)
    @variable(sb, z[j=N,t=T] >= 0)
    @objective(sb, Min,
    sum(c[i,j,t,s]*y[i,j,t] for i in R for j in N for t in T)
    + sum(c0[j,t,s]*z[j,t] for j in N for t in T)
    )
    @constraint(sb, [i=R, t=T],
    -sum(x[i,tau] for tau in 1:t)
    + sum(d[j,t,s]*y[i,j,t] for j in N) <= 0)
    @constraint(sb, [j=N, t=T],
    sum(y[i,j,t] for i in R) + z[j,t] == 1
    )
    end

    return model

end
```

Figure 4: Example: `dcap_models.jl`

1.2.2 `Julia` scripts for convenient functionalities

To provide convenience for users, we implement various functions which are hopefully useful in terms of investigating SIP instances. The following `Julia` files stand for that purpose.

`smpswriter.jl` This script implements the basic building blocks for the functions that converts `JuMP.Model` object to SMPS files.

`generator.jl` This script defines interfacing functions for generating SMPS files as well as constructing `JuMP.Model` object.

`analyzer.jl` This script implements the functions for analysis of instances. This includes getting size information, sparsity information, and plots of sparsity patterns.

`utility.jl` This script contains some utility functions that help implementing the package.

2 Tutorial

2.1 Prerequisites

We assume that you are in Linux environment. To use `Siplib.jl`, you need to perform the following steps:

1. install the latest `Julia` release.
2. install `Julia` packages `Distributions.jl`, `StructJuMP.jl`, `PyPlot.jl` by executing
   - `Pkg.add("Distributions")`
   - `Pkg.add("StructJuMP")`
   - `Pkg.add("PyPlot")`
3. download and place the `Siplib.jl` package to any directory in your computer
4. open a terminal and change working directory to "`any-directory-in-your-computer/Siplib/src/`"
5. run `Julia` in that directory
6. excute `include("Siplib.jl")`
7. excute `using Siplib`

Then, you are all set to use the functions in `Siplib.jl`.

2.2 Generating instances

`Siplib.jl` provides four functions with regard to instance generation:

- `getInstanceName()`
- `getModel()`
- `writeSMPS()`
- `generateSMPS()`

In short, `getInstanceName()` returns a `String`-type instance name, `getModel()` constructs `JuMP.Model`-type object, `writeSMPS()` converts a `JuMP.Model`-type object to the three SMPS files, and `generateSMPS()` does both of them simultaneously.

2.2.1 function `getInstanceName()`

```
function getInstanceName(problem::Symbol, params_arr::Any)::String
```

The function `getInstanceName()` is an utility function that returns a `String`-type instance name defined in Table 2. It takes two necessary input arguments `problem` and `params_arr`:

> `problem` (necessary, positional) The `Symbol`-type argument that specify the problem of which we want to generate instance. The appropriate values are given in Table 3.
>
> `params_arr` (necessary, positional) The argument that specifies the parameters of the problem. It must be properly paired with the argument `problem`. The appropriate values are given in Table 3.

Table 3: Acceptable values for `problem` and `params_arr` arguments pairs

problem	params_arr	Remark
:DCAP	[R, T, N, $\mathcal{S}$]	All parameters are integer.
:MPTSPs	[D, N, $\mathcal{S}$]	String D $\in$ {"D0", "D1", "D2", "D3"}. All other parameters are integer.
:SIZES	[$\mathcal{S}$]	Integer $\mathcal{S} \geq 20$.
:SMKP	[I, $\mathcal{S}$]	All parameters are integer.
:SSLP	[I, J, $\mathcal{S}$]	All parameters are integer.
:SUC	[D, $\mathcal{S}$]	String D $\in$ {"FallWD", "FallWE", "WinterWD", "WinterWE", "SpringWD", "SpringWE", "SummerWD", "SummerWE"}. $\mathcal{S}$ is integer.

2.2.2 function `getModel()`

```
function getModel(problem::Symbol, params_arr::Any ; seed::Int=1, lprelax::Int=0)::
    JuMP.Model
```

The function `getModel()` returns a `JuMP.Model`-type object. It has two necessary positional arguments and two optional keyword arguments:

> `problem` (necessary, positional) The `Symbol`-type argument that specify the problem of which we want to generate instance. The appropriate values are given in Table 3.
>
> `params_arr` (necessary, positional) The argument that specifies the parameters of the problem. It must be properly paired with the argument `problem`. The appropriate values are given in Table 3.
>
> `seed` (optional, keyword) The integer argument `seed` specifies the seed of pseudo-random number generator in `Julia`. If specific value is not supplied, `seed=1` as a default.

`lprelax` (optional, keyword) The keyword argument specifying the level of LP-relaxation of an instance. Table 4 summarizes the acceptable values with its meaning. If not specified, `lprelax=0` as a default which means no LP-relaxation.

Table 4: Acceptable values for `lprelax` argument

`lprelax`	Remark
0 (default)	No LP-relaxation
1	First-stage only LP-relaxation
2	Second-stage only LP-relaxation
3	Full LP-relaxation (both first-stage and second-stage)

For example, executing the following line constructs a `JuMP.Model` object `model` of instance DCAP_3_4_2_100 with default random seed 1 and without LP-relaxation.

```
julia> model = getModel(:DCAP, [3,4,2,100])
```

The keyword argument `seed` can be changed by another value, for example,

```
julia> model = getModel(:DCAP, [3,4,2,100], seed=2)
```

The line below constructs a `JuMP.Model` object of the second-stage only LP-relaxed instance DCAP_3_4_2_100 using a different random seed 2.

```
julia> model = getModel(:DCAP, [3,4,2,100], seed=2, lprelax=2)
```

2.2.3 function `writeSMPS()`

```
function writeSMPS(model::JuMP.Model, INSTANCE_NAME::String="instance", DIR_NAME::
    String="$(dirname(@__FILE__))/../instance"; genericnames::Bool=true, splice::Bool
    =true)
```

The function `writeSMPS()` converts a `JuMP.Model` object to SMPS files. It takes up to five inputs:

`model` (necessary, positional) The `JuMP.Model`-type object of which we want to generate SMPS files.

`INSTANCE_NAME` (optional, positional) The `String`-type argument that will be the name of SMPS files. If not specified, `INSTANCE_NAME="instance"` as a default. Then, the three files are generated:

- instance.cor
- instance.tim
- instance.sto

DIR_NAME (optional, positional) The `String`-type argument to indicate a directory where the files are stored. The SMPS files are stored in the default folder "~/Siplib/instance/" unless the argument `DIR_NAME` is specified.

`genericnames` (optional, keyword) The `Bool`-type keyword argument that decides whether or not we keep the original variable names.

If `genericnames=true`, SMPS files are written with the variable names `VAR1`, `VAR2`, `VAR3`, and so on. If `genericnames=false`, the original variable names such as `x[1,1]`, `y[2,2]`, and `z[3,3]` are maintained. `true` as a default.

`splice` (optional, keyword) The `Bool`-type keyword argument that decides whether or not the stochastic data stored in the `JuMP.Model`-type object is spliced after writing the SMPS files. Hence, it increases the memory efficiency during the generation of SMPS files. However, the spliced `JuMP.Model`-type object cannot be re-used for further purpose. `true` as a default.

Executing the following lines store the three SMPS files of DCAP_3_4_2_100 in the default directory "~/Siplib/instance/" with the default file name "instance", generic variable names, and splicing stored data.

```
julia> model = getModel(:DCAP, [3,4,2,100])
julia> writeSMPS(model)
```

The optional inputs can be replaced like below.

```
julia> model = getModel(:DCAP, [3,4,2,100])
julia> writeSMPS(model, "DCAP_3_4_2_100", "/another/directory", genericnames=false,
     splice=false)
```

The above lines store SMPS files named by DCAP_3_4_2_100.cor, DCAP_3_4_2_100.tim, DCAP_3_4_2_100.sto into the directory "/another/directory" with the original variable names `x`, `y`, `u` and without splicing the data in the object `model`.

2.2.4 function `generateSMPS()`

```
function generateSMPS(problem::Symbol, params_arr::Any, DIR_NAME::String="$(dirname(
    @__FILE__))/../instance" ; seed::Int=1, lprelax::Int=0, genericnames::Bool=true,
    splice::Bool=true)
```

`generateSMPS()` generates SMPS files as well as returns `JuMP.Model` object. It is simply the combination of the two functions: `getModel()` and `writeSMPS()`.

One benefit of using this function is its functionality to generate instance name automatically using the input arguments. For example, the following line returns `JuMP.Model`-type object `model` as well as generates three SMPS files.

```
julia> model = generateSMPS(:DCAP, [3,4,2,100], splice=false)
```

`generateSMPS()` takes up to seven argument inputs:

problem (necessary, positional) The `Symbol`-type argument that specify the problem of which we want to generate instance. The appropriate values are given in Table 3.

params_arr (necessary, positional) The array argument that specifies the parameters of the problem. It must be properly paired with the argument `problem`. The appropriate values are given in Table 3.

DIR_NAME (optional, positional) The `String`-type argument to indicate a directory where the files are stored. The SMPS files are stored in the default folder "∼/Siplib/instance/" unless the argument `DIR_NAME` is specified.

seed (optional, keyword) The integer argument `seed` specifies the seed of pseudo-random number generator in `Julia`. If specific value is not supplied, `seed=1` as a default.

lprelax (optional, keyword) The keyword argument specifying the level of LP-relaxation of an instance. Table 4 summarizes the acceptable values with its meaning. If not specified, `lprelax=0` as a default which means no LP-relaxation.

Noticeably, the SMPS files are named to denote the level of LP-relaxation unless `lprelax=0`. For example, setting `lprelax=2` for this function with a pair `problem=:DCAP` and `params_arr=[3,4,2,100]` generates three SMPS files:

- DCAP_3_4_2_100_LP2.cor
- DCAP_3_4_2_100_LP2.tim
- DCAP_3_4_2_100_LP2.sto

genericnames (optional, keyword) The `Bool`-type keyword argument that decides whether or not we keep the original variable names.

If `genericnames=true`, SMPS files are written with the variable names `VAR1`, `VAR2`, `VAR3`, and so on. If `genericnames=false`, the original variable names such as `x[1,1]`, `y[2,2]`, and `z[3,3]` are maintained. `true` as a default.

splice (optional, keyword) The `Bool`-type keyword argument that decides whether or not the stochastic data stored in the `JuMP.Model`-type object is spliced after writing the SMPS files. Hence, it increases the memory efficiency during the generation of SMPS files. However, the spliced `JuMP.Model`-type object cannot be re-used for further purpose. `true` as a default.

2.3 Pre-analyzing instances: size, sparsity, plot

`Siplib.jl` provides pre-analysis functions for instances. By "size", we mean the number of components (continuous, binary, integer, constraint) in an instance. The sparsity is analyzed block-wisely. The size and sparsity information is stored in the object of the following composite types: `Size` and `Sparsity`.

`Siplib.jl` also provides functions to plot sparsity pattern in the constraint matrix. The types of plot that can be drawn are:

- Constraint matrix of extensive form
- First-stage block (block A)
- Second-stage block (block W)
- Technology block (block T)

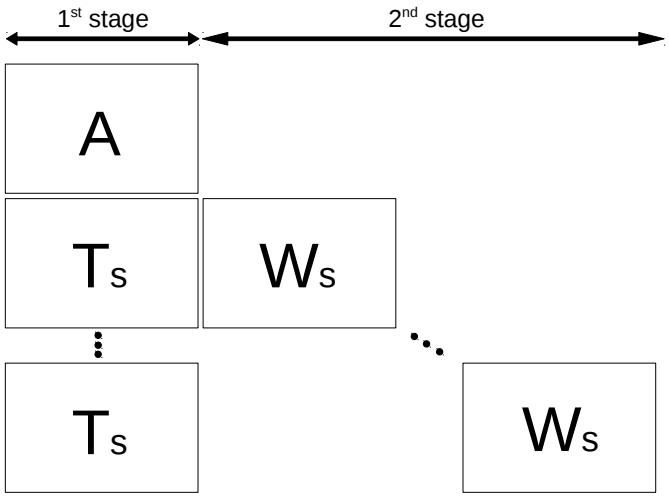


Figure 5: Three independent blocks in SIP

2.3.1 function `getSize()`

To get the size information of an instance, use `getSize()` with one of the following input arguments set:

```
function getSize(problem::Symbol, params_arr::Any)::Size
function getSize(model::JuMP.Model, INSTANCE_NAME::String="")::Size
```

Then, it returns `Size`-type object defined as follows.

```
mutable struct Size
    INSTANCE_NAME::String # instance name
    nCont1::Int # number of continuous variables in 1st stage
    nBin1::Int # number of binary variables in 1st stage
    nInt1::Int # number of integer variables in 1st stage
    nCont2::Int # number of continuous variables in 2nd stage
    nBin2::Int # number of binary variables in 2nd stage
    nInt2::Int # number of integer variables in 2nd stage
    nCont::Int # number of continuous variables in total
    nBin::Int # number of binary variables in total
    nInt::Int # number of integer variables in total
    nRow::Int # number of rows in coefficient matrix in extensive form
    nCol::Int # number of columns in coefficient matrix in extensive form
    nNz::Int # number of nonzero values in coefficient matrix in extensive form
    Size() = new()
end
```

2.3.2 function `getSparsity()`

To get the sparsity information of an instance, excute the following function.

```
function getSparsity(problem::Symbol, params_arr::Any)::Sparsity
function getSparsity(model::JuMP.Model, INSTANCE_NAME::String="")::Sparsity
```

Then, it returns `Sparsity`-type object defined as follows.

```
mutable struct Sparsity
    INSTANCE_NAME::String # instance name
    nRow1::Int # number of rows in 1st stage−only block (block A)
    nCol1::Int # number of columns in 1st stage−only block (block A)
    nNz1::Int # number of nonzero values in 1st stage−only block (block A)
    sparsity1::Float64 # sparsity ([0,1] scale) of 1st stage−only block (block A)
    nRow2::Int # number of rows in 2nd stage−only block (block W)
    nCol2::Int # number of columns in 2nd stage−only block (block W)
    nNz2::Int # number of nonzero values in 2nd stage−only block (block W)
    sparsity2::Float64 # sparsity ([0,1] scale) of 2nd stage−only block (block W)
    nRowC::Int # number of rows in technology block (block T)
    nColC::Int # number of columns in technology block (block T)
    nNzC::Int # number of nonzero values in technology block (block T)
    sparsityC::Float64 # sparsity ([0,1] scale) of technology block (block T)
    nRow::Int # number of rows in total
    nCol::Int # number of columns in total
    nNz::Int # number of nonzero values in total
    sparsity::Float64 # sparsity ([0,1] scale) in total
    Sparsity() = new()
end
```

2.3.3 Five functions for plotting sparsity pattern

To plot the sparsity patterns of coefficient matrices, we provide the following functions.

```
function plotConstrMatrix(model::JuMP.Model, INSTANCE_NAME::String="instance",
    DIR_NAME::String="$(dirname(@__FILE__))/../plot")

function plotFirstStageBlock(model::JuMP.Model, INSTANCE_NAME::String="
    instance_block_A", DIR_NAME::String="$(dirname(@__FILE__))/../plot")

function plotSecondStageBlock(model::JuMP.Model, INSTANCE_NAME::String="
    instance_block_W", DIR_NAME::String="$(dirname(@__FILE__))/../plot")

function plotTechnologyBlock(model::JuMP.Model, INSTANCE_NAME::String="
    instance_block_T", DIR_NAME::String="$(dirname(@__FILE__))/../plot")

function plotAllBlocks(model::JuMP.Model, INSTANCE_NAME::String="instance", DIR_NAME::
    String="$(dirname(@__FILE__))/../plot")

function plotAll(model::JuMP.Model, INSTANCE_NAME::String="instance", DIR_NAME::String
    ="$(dirname(@__FILE__))/../plot")
```

All the functions above take up to three input arguments:

`model` (necessary, positional) The `JuMP.Model`-type object of which we want to draw plots.

`INSTANCE_NAME` (optional, positional) The `String`-type argument that will be the name of plot files. If not specified, `INSTANCE_NAME="instance_"` as a default. Plots are stored in .pdf format.

`DIR_NAME` (optional, positional) The `String`-type argument to indicate a directory where the files are stored. The .pdf file is stored in the default folder "`~/Siplib/plot/`" unless the argument `DIR_NAME` is specified.

The function `plotConstrMatrix` plots the whole constraint matrix of extensive form. For example, the following command lines plot Fig. 6b.

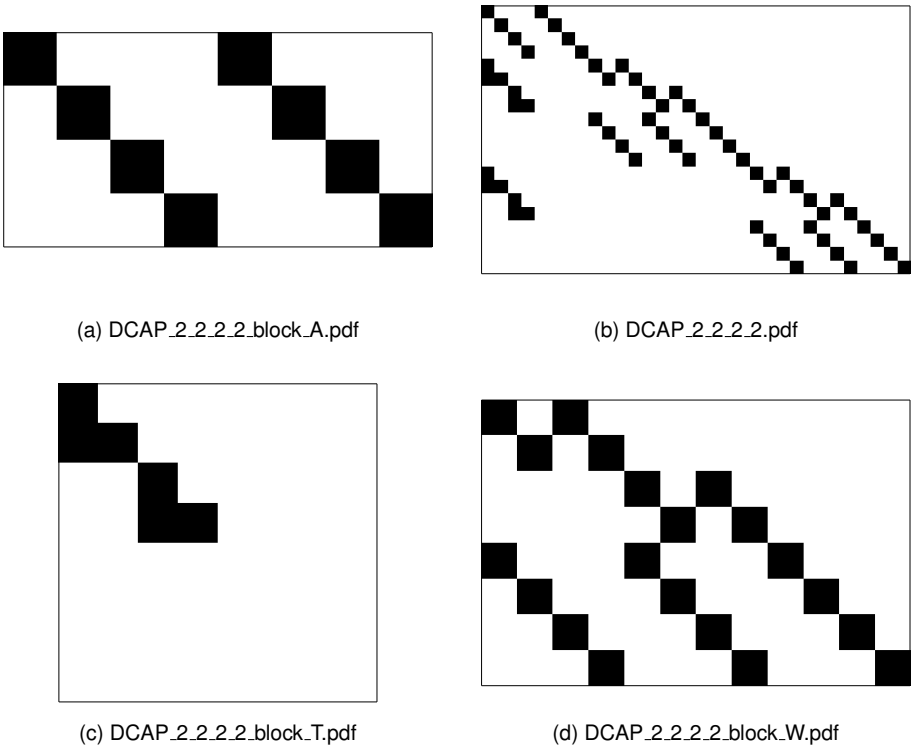
```
params_arr = [2,2,2,2] # declare parameters
problem = :DCAP # declare problem
INSTANCE_NAME = getInstanceName(problem, params_arr) # get instance name
model = getModel(problem, params_arr) # construct JuMP.Model object
plotConstrMatrix(model, INSTANCE_NAME) # plot extensive form constraint matrix
```

The functions `plotFirstStageBlock()`, `plotSecondStageBlock()`, and `plotTechnologyBlock()` all take `JuMP.Model`-type object and plots each block. For example, the following command lines plot Fig. 6a, 6c, and 6d.

```
params_arr = [2,2,2,2] # declare parameters
problem = :DCAP # declare problem
INSTANCE_NAME = getInstanceName(problem, params_arr) # save instance name
model = getModel(problem, params_arr) # construct JuMP.Model object
plotFirstStageBlock(model, INSTANCE_NAME) # plot 1st stage block
plotSecondStageBlock(model, INSTANCE_NAME) # plot 2nd stage block
plotTechnologyBlock(model, INSTANCE_NAME) # plot technology block
```

One might want to draw all the plots at once. The following two functions are defined to do that.

```
plotAllBlocks(model, INSTANCE_NAME) # plot all blocks A, W, and T, respectively
plotAll(model, INSTANCE_NAME) # plot all the plots above: EF constraint matrix and blocks A, W, T
```



(a) DCAP_2_2_2_2_block_A.pdf (b) DCAP_2_2_2_2.pdf

(c) DCAP_2_2_2_2_block_T.pdf (d) DCAP_2_2_2_2_block_W.pdf

Figure 6: Plots drawn by executing function `plotAll`

By executing `plotAll()`, one can obtain all the plots in Fig. 6.

References

[1] S. Ahmed, R. Garcia, N. Kong, L. Ntaimo, G. Parija, F. Qiu, S. Sen. SIPLIB: A Stochastic Integer Programming Test Problem Library. http://www.isye.gatech.edu/ sahmed/siplib, 2015.

[2] S. Ahmed and R. Garcia. "Dynamic Capacity Acquisition and Assignment under Uncertainty," Annals of Operations Research, vol.124, pp. 267-283, 2003.

[3] R. Tadei, G. Perboli, and F. Perfetti, The multi-path traveling salesman problem with stochastic travel costs, EURO Journal on Transportation and Logistics, 2017

[4] Soheila Jorjani, Carlton H. Scott, and David L. Woodruff, Selection of an optimal subset of sizes, International Journal of Production Research, 1999

[5] Gustavo Angulo, Shabbir Ahmed, and Santanu S. Dey, Improving the integer L-shaped method, 2014.

[6] Anthony Papavasiliou and Shmuel S. Oren, Multiarea stochastic unit commitment for high wind penetration in a transmission constrained network, Operations Research, 2013

[7] Lewis Ntaimo and Suvrajeet Sen, The million-variable "March" for stochastic combinatorial optimization, Journal of Global Optimization, 2005