Modeling decomposable Mixed Integer Programs

JuMP-dev Workshop 2018, Bordeaux

Guillaume Marques, Vitor Nesello, Artur Pessoa, Ruslan Sadykov, Issam Tahiri, François Vanderbeck

Université de Bordeaux, Inria, Universidade Federal Fluminense
Plan

- BlockDecomposition.jl - Modeling
- BlockDecomposition.jl - Pricing Callbacks
- RCSP.jl - Pricing Callback Generator
- Supported solvers

In following slides

```julia
const BD = BlockDecomposition
const RM = RCSP.Modeling
const RS = RCSP.Solver
```
BlockDecomposition.jl

Modeling
We partition constraints.

- Constraints $mc_1$ to $mc_m$ are in the master.
- Constraints $sc_{1,1}$ to $sc_{1,o}$ are in the 1st subproblem.
- Constraints $sc_{2,1}$ to $sc_{2,p}$ are in the 2nd subproblem.
- Constraints $sc_{3,1}$ to $sc_{3,q}$ are in the 3rd subproblem.

A function to describe this decomposition

```deshdecomp(constr_name, constr_id)
if constr_name == :mc
  return (:DW_MASTER, 0)
else
  return (:DW_SP, constr_id[1])
end
end
```

BD.add_dantzig_wolfe_decomposition(m, dw_decomp)
We partition variables.

- Variables $y_\alpha$, $\alpha \in 1 \ldots h$ are in the master.
- Variables $x_{1,\alpha}$, $\alpha \in 1 \ldots i$ are in the 1st subproblem.
- Variables $x_{2,\alpha}$, $\alpha \in 1 \ldots j$ are in the 2nd subproblem.
- Variables $x_{3,\alpha}$, $\alpha \in 1 \ldots k$ are in the 3rd subproblem.

A function to describe this decomposition

```python
function b_decomp(var_name, var_id)
    if var_name == :y
        return (:B_MASTER, 0)
    else
        return (:B_SP, var_id[1])
    end
end
BD.add_benders_decomposition(m, b_decomp)
```
Generalized Assignment Problem

Assign each job to a machine at minimum cost while not exceeding capacities of machines.

Let $x_{mj}$ equals 1 if job $j$ is assigned to machine $m$; 0 otherwise.

```plaintext
gap = Model(solver = Solver())
@variable(gap, x[m in Machines, j in Jobs], Bin)
@constraint(gap, cov[j in Jobs],
    sum( x[m,j], m in Machines ) >= 1)
@constraint(gap, knp[m in Machines],
    sum(weight[m,j]*x[m,j], j in Jobs) <= capacity[m])
@objective(gap, Min,
    sum(cost[m,j]*x[m,j], m in Machines, j in Jobs))
solve(gap)
```
Decomp.

Dantzig-Wolfe

Benders

Example

gap = BD.BlockModel(solver = BaPCodSolver())

@variable(gap, x[m in Machines, j in Jobs], Bin)

@constraint(gap, cov[j in Jobs],
    sum(x[m,j], m in Machines) >= 1)

@constraint(gap, knp[m in Machines],
    sum(weight[m,j]*x[m,j], j in Jobs) <= capacity[m])

@objective(gap, Min,
    sum(cost[m,j]*x[m,j], m in Machines, j in Jobs))

function dw_fct(cstr_name, cstr_id)
    if cstr_name == :cov  # cov constraints are assigned to
        return (:DW_MASTER, 1)  # the master that has the index 1
    else
        return (:DW_SP, cstr_id)  # knp constraints are assigned to
    end
end

BD.add_dantzig_wolfe_decomposition(gap, dw_fct)
BlockDecomposition.jl

Pricing Callbacks
Pricing callbacks can be used to solve efficiently subproblems.

Available functions:

```
function BD.getcurcost(cb, var)::Float64
function BD.getcurub(cb, var)::Float64
function BD.getcurlb(cb, var)::Float64
function BD.setsolutionvalue(cb, var, value)::Void
```

We introduce them with the Generalized Assignment Problem.
A function solving efficiently the knapsack problem.

\[
(sol, value) = \text{solveKnp}(\text{costs}, \text{weights}, \text{capacity})
\]

A pricing callback using this function:

```python
function myKnapsackSolver(cb)
    machine = BD.getspid(cb)[1]  # machine index

    costs = [BD.getcurcost(x[machine, j]) for j in Jobs]

    (sol_x_m, value) = solveKnp(costs, Weight[m,:], Capacity[m])

    for j in data.jobs
        BD.setsolutionvalue(cb, x[machine, j], sol_x_m[j])
    end
end
```
Pricing callback

Definition

Example

gap = BD.BlockModel(solver = BaPCodSolver())

@variable(gap, x[m in Machines, j in Jobs], Bin)

@constraint(gap, cov[j in Jobs],
            sum(x[m,j], m in Machines) >= 1)

@constraint(gap, knp[m in Machines],
            sum(weight[m,j]*x[m,j], j in Jobs) <= capacity[m])

@objective(gap, Min,
            sum(cost[m,j]*x[m,j], m in Machines, j in Jobs))

function dw_fct(cstr_name, cstr_id)
    if cstr_name == :cov
        return (:DW_MASTER, 1)
    else
        return (:DW_SP, cstr_id)
    end
end

BD.add_dantzig_wolfe_decomposition(gap, dw_fct)

# Pricing callback assignment
for m in Machines
    BD.addpricingcbtosp!(gap, m, myKnapsackSolver)
end
Pricing callback

Definition

Example

gap = BD.BlockModel(solver = BaPCodSolver())

@variable(gap, x[m in Machines, j in Jobs], Bin)

@constraint(gap, cov[j in Jobs],
            sum(x[m,j], m in Machines) >= 1)

@objective(gap, Min, 
            sum(cost[m,j]*x[m,j], m in Machines, j in Jobs))

# Decomposition on constraints
dw_fct(cstr_name, cstr_id) = (:DW_MASTER, 1)
BD.add_dantzig_wolfe_decomposition(gap, dw_fct)

# Decomposition on variables
dw_fct_on_vars(var_name, var_id) = (:DW_SP, var_id[1])
BD.add_dantzig_wolfe_decomposition_on_variables(gap, dw_fct_on_vars)

# Pricing callback assignment
for m in Machines
    BD.addpricingcbtosp!(gap, m, myKnapsackSolver)
end
RCSP.jl

Resource Constrained Shortest Path Pricing Callback Generator
RCSP

Definition

- Structure of the network
- Variables to edges assignment
- Edges / Vertices resources properties
  - consumption and bounds

Figure: Example of RCSP feasible solution
Heterogeneous Vehicle Routing Problem With Time Windows (HVRPTW) formulation:

\[
\min \sum_{k=1}^{U} \sum_{i,j} c_{ij}^k x_{ij}^k
\]

s.t. \[ \sum_{k=1}^{U} \sum_{i,j} x_{ij}^k = 2 \quad j \in V \setminus \{depot\} \]

\[ x^k \in X^k \quad k = 1..U \]

- \( x_{ij}^k = 1 \) if edge \((i, j)\) is used by vehicle \(k\)
- \( c_{ij}^k \) cost of edge \((i, j)\) for vehicle \(k\)
- \( U \) number of heterogeneous vehicles
- \( X^k \) set of tours visiting a subset of customers within their time windows that vehicle \(k\) can do.

Tours \(X^k\) are generated for each vehicle \(k\) by a pricing callback.
Compact formulation + decomposition functions.

```python
vrp = BD.BlockModel(solver = BaPCodSolver())

@variable(vrp, x[k in K, a in Arcs], Int)

@constraint(vrp, part[c in C],
    sum(x[k, a] for k in K, a in incident_arcs(c)) == 2.0)

@objective(vrp, Min, sum(cost(k, a) * x[k, a] for k in K, a in Arcs))

# Decomposition on constraints
dw(cstr_name, cstr_id) = (:DW_MASTER, 0)
BD.add_dantzig_wolfe_decomposition(vrp, dw)

# Decomposition on variables
dw_on_vars(var_name, var_id) = (:DW_SP, var_id[1])
BD.add_dantzig_wolfe_decomposition_on_variables(vrp, dw_on_vars)
```
For a given vehicle $k$, tours are generated solving a RCSP.

- Variable $x_{ij}^k$ is assigned to edge $(i, j)$
- Resource is time
- Resource consumption on edge is travel time of vehicle $k$.
- Bounds on accumulated resource consumption at vertices are time windows.
Network is the road network.

```python
network = RM.Network(nb_nodes, source = 1, sink = nb_customers + 2)
```

Resource is time.

```python
time_res = RM.addresource!(network)
```

Definition of time windows.

```python
for v in Vertices
    RM.setresourceproperties!(network, v, time_res, lb = a(v), ub = b(v))
end
```
Instantiation of edges.

```python
for c1 in Customers, c2 in Customers
    if c1 != c2
        edge = RM.add_edge!(network, (c1, c2), var = x[k, (c1, c2)])
        RM.setresourceproperties!(network, edge, time_res,
                                  consumption = traveltime(k, c1, c2))
    end
end

for c in Customers
    # Source
    edge = RM.add_edge!(network, depot, c, var = x[k, (depot, c)])
    RM.setresourceproperties!(network, edge, time_res,
                              consumption = traveltime(k, depot, c))

    # Sink
    edge = RM.add_edge!(network, c, sink, var = x[k, (depot, c)])
    RM.setresourceproperties!(network, edge, time_res,
                              consumption = traveltime(k, c, sink))
end
```
A function wrapping the definition of the RCSP problem.

```python
function vrptw_rcsp(cb)
    k = BD.getspid(cb)[1]  # Get the vehicle id
    network = RM.Network(nb_customers + 2)
    # Define the network, resource, etc.
    return network
end
```

Generation of a pricing callback for each subproblem.

```python
for k in K
    RS.generate_rcsp_callback!(vrp, (:DW_SP, k), vrptw_rcsp)
end
```

Multiplicity equals the number of vehicles of each type.

```python
BD.addspmultiplicity(vrp, (spid, sptype) -> (0, 1))
```
Supported solvers
**CPLEX**

- **BlockDecomposition.jl** (Benders only)
- **JuMP.jl**
- **MathOptInterface.jl**
- **CPLEX.jl**

**Cplex**

- `CPXPARAM_Benders_Strategy`
  - `-1` Solves Compact formulation
  - `1` Solves Benders formulation

**Julia**

**C++**
Coluna.jl (ongoing work)
Thank you!

Questions?