Power Systems Optimization

Group 2

Tomás Valencia Zuluaga   Tim Schmidtlein   Noranne Gabouge

May 13, 2020

Abstract

We apply *Benders' decomposition* to a Unit Commitment and Economic Dispatch problem with Security Constraints (SC-UCED), formulated as a MIP. We model costs as a piecewise linear convex function and assume fast decoupled power flow without losses. The original problem formulation is decomposed into three layers, each one of them corresponding to one specific component of the problem (UC master problem, ED subproblems, SC sub-subproblems). We conduct computational experiments and compare the results to the performance of an off-the-shelf solver.

1 Problem description and scope

1.1 The SC-UCED problem

The problem we address is the very important and common problem of Unit Commitment and Economic Dispatch, which is fundamental in the operation of electrical grids. The setting of our problem is the following: before the beginning of an operation period, the power system operator receives the demand forecast in each time period and price offers of each generating unit in the system for the entire operation horizon. We consider here an operation period of one day and time intervals of one hour. With this information, the operator must solve the following two problems:

- The *Unit Commitment* problem (UC) consists in selecting which generating units will be turned on and which ones will be left unused, in each time period. The operator must strive here to minimize fixed operation cost while committing enough generators to supply all demand, satisfy operational requirements such as spinning reserves and minimum running times and satisfy system requirements, which amounts to guaranteeing that the Economic Dispatch problem will be feasible.

- The *Economic Dispatch* problem (ED) consists in determining the output of each generating unit that has been selected in the UC. The constraints that must be satisfied here are the physical constraints of the electric system: mainly verifying that transmission line capacities will not be exceeded in the base case or in any of a select set of contingencies. In this stage, the operator aims at minimizing the variable cost of generating electricity.

Since we are not only considering the operation of the system in its normal state but also in the case that one of a select set of contingencies occurs, we speak of a *Security-Constrained Unit Commitment and Economic Dispatch* (SC-UCED).

1.2 Scope

This problem becomes increasingly complicated as more periods and scenarios are taken into consideration. For the scope of this paper, we are limiting our approach to solving the SC-UCED problem for one day with hourly intervals (half-hourly intervals for some computational experiments), applying *Benders' decomposition* and relying on a simplified *linear* formulation of power flow (so-called DC Power Flow), modeled on the example of the work found in [1]. Moreover, we do not consider operational requirements in the UC stage of the problem beyond maintaining ED feasibility. We consider fixed costs corresponding to both startup and shutdown operations. For variable costs, we choose a piecewise linear convex cost function for the generators (in contrast to the quadratic convex cost in [1]) to be able to apply the Benders' cut generation algorithm. Solving the problem for a nonlinear convex cost function relies on the *Generalized Benders' decomposition*, an extension of Benders' decomposition to nonlinear programs. In the center of our interest stands the question of the value *Benders' decomposition* adds to solving the problem in terms of computation in a piecwise linear setting, compared to solving the original problem.

2 Model formulation

In this section, we define variables, parameters and constraints for the SC-UCED problem, and formulate the problem as a Mixed-Integer Program.

2.1 Parameters and decision variables

Sets

$\mathcal{G}$	Set of generators
$\mathcal{N}$	Set of busses
$\mathcal{B}$	Set of branches (transformers and transimssion lines)
$\mathcal{T}$	Set of time periods in the planning horizon
$\mathcal{L}$	Set of linear segments into which the cost function is broken down
$\mathcal{S}$	Set of base case and contingency scenarios

Parameters

Generator parameters

$P_{g,l}^{max}$	Maximal output of segment l of generator g
P_g^{min}	Minimal output of generator g
C_g^{min}	Cost of operating generator g at minimum output
$C_{g,l}$	Slope of segment l of piecewise linear cost function of generator g (see Figure 1).
SU_g	Startup cost of generator g
SD_g	Shutdown cost of generator g

For reference, consider the piecewise linear cost function below in Figure 1.

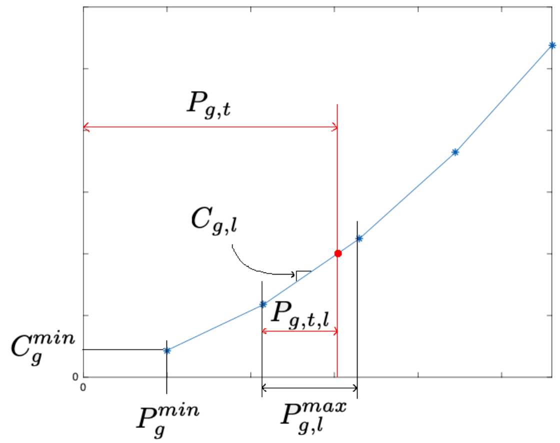


Figure 1: Piecewise linear cost function. In the picture, $L = 4$ and parameters are shown for segment $l = 2$. The red dot represents the generator output for the current period t.

System parameters

$P_{i,t}^L$	Demand at bus i during period t
$\overrightarrow{P_t^L}$	Vector in $\mathbb{R}^{\lvert\mathcal{N}\rvert}$ with i-th component equal to $P_{i,t}^L$
DF_s	Distribution factor matrix of the system in scenario s, defined below.
V	$\lvert\mathcal{N}\rvert$-by-$\lvert\mathcal{G}\rvert$ connection matrix. $V(i,g)$ is 1 if generator g is connected to bus i, 0 otherwise.
ϕ_b^{max}	Maximal power flow in branch b
$\overrightarrow{\phi^{max}}$	Vector in $\mathbb{R}^{\lvert\mathcal{B}\rvert}$ with b-th component equal to ϕ_b^{max}. (Same parameter as above, in vector form).

DF_s is the distribution factor matrix, a $|\mathcal{B}|$-by-$|\mathcal{N}|$ matrix which, when multiplied with the vector of net power injections, yields the flow in each branch of the system [3]. The vector of net power injections is $V \cdot \overrightarrow{P_t} - \overrightarrow{P_t^L}$

Main decision variables

UC Decision variables	
$u_{g,t}$	Commitment decision for generator g during period t. 1 if turned on, 0 otherwise.
ED Decision variables	
$P_{g,t}$	Output of generator g during period t (in MWh).
$\overrightarrow{P_t}$	Vector in $\mathbb{R}^{\lvert\mathcal{G}\rvert}$, with g-th component equal to $P_{g,t}$. (Same decision variable as above, in vector form).

Auxiliary decision variables

We call auxiliary decision variables those decision variables in the mathematical model that are added for ease of modelling, but are completely determined by the main decision variables by equality or inequality constraints.

UC Decision variables	
$SD_{g,t}$	Shutdown cost of generator g during period t
$SU_{g,t}$	Startup cost of generator g during period t
ED Decision variables	
$P_{g,t,l}$	Segment l of the output of generator g during period t

2.2 Constraints

Unit commitment constraints
Shutdown and startup costs:

$$SD_{g,t} \geq SD_g(u_{g,t-1} - u_{g,t}) \ \ \forall g \forall t \tag{1}$$

$$SU_{g,t} \geq SU_g(u_{g,t} - u_{g,t-1}) \ \ \forall g \forall t \tag{2}$$

System constraints (Economic Dispatch constraints)
Total generator output equal to sum of linear segments:

$$P_{g,t} = P_g^{min} + \sum_{l \in L} P_{g,t,l}, \ \ \forall g, \forall t \tag{3}$$

Power balance:

$$\sum_{g \in \mathcal{G}} P_{g,t} = \sum_{i \in \mathcal{N}} P_{i,t}^L \ \ \forall t \tag{4}$$

Transmission line power flow constraints:

$$-\overrightarrow{\phi^{max}} + DF_s \overrightarrow{P_t^L} \leq DF_s \cdot V \cdot \overrightarrow{P_t} \leq \overrightarrow{\phi^{max}} + DF_s \overrightarrow{P_t^L}, \ \ \forall s \forall t \tag{5}$$

Coupling constraints between UC and ED:

$$0 \leq P_{g,t,l} \leq u_{g,t} P_{g,l}^{max}, \ \ \forall g, \forall l, \forall t \tag{6}$$

2.3 Model

$$\begin{aligned}
\underset{u,P}{\text{minimize}} \quad & \sum_{t \in \mathcal{T}} \sum_{g \in \mathcal{G}} \left(u_{g,t} C_g^{min} + \sum_{l \in \mathcal{L}} (C_{g,l} P_{g,t,l}) + SU_{g,t} + SD_{g,t} \right) \\
\text{subject to} \quad & \text{UC constraints (1)-(2),} \\
& \text{ED constraints (3)-(5),} \\
& \text{Coupling constraints (6),} \\
& u_{g,t} \in \{0,1\} \ \ \forall g \forall t, \\
& SU_{g,t}, SD_{g,t} \in \mathbb{R}_+, \ \ \forall g \forall t, \\
& P_{g,t}, P_{g,t,l} \in \mathbb{R}_+ \ \ \forall g \forall l \forall t
\end{aligned}$$

3 Methodology

In the model above, the problems of Unit Commitment and Economic Dispatch are solved simultaneously, which may quickly become computationally challenging for large instances. We are therefore going to apply the algorithm of *Benders' decomposition* to split the optimization of MIP into smaller subproblems with the goal of reducing overall computation time. *Benders' decomposition* generally enjoys great popularity in the field of Energy Systems Optimization, for instance in generation expansion problems [4] and for problems in a Two-Stage Stochastic Programming setting [5].

3.1 The method of *Benders' decomposition*

The *Benders' decomposition* algorithm itself relies on *Benders' reformulation* [6], a way of rewriting a MIP that has multiple complicating variables as an equivalent problem that has only a single complicating variable (in addition to the non-complicating variables). More precisely, for *Benders' reformulation*, the continuous variables are seen as the complicating variables to an otherwise pure integer program. Given a generic MIP of the form:

$$\begin{aligned} \textbf{(MIP)} \quad z_{IP} = \max \quad & c^T x + h^T y \\ \text{s.t.} \, & Ax + Gy \leq b \\ & x \in \mathbb{Z}^n_+, y \in \mathbb{R}^p_+ \end{aligned}$$

we can obtain the following equivalent problem via *Benders' reformulation*:

$$\begin{aligned} \textbf{(MIP')} \qquad & z_{\text{IP}} = \min \ cx + \mu \\ & \text{s.t.} \qquad \mu \leq u^k \cdot (b - Ax), \quad \text{for all } k \in K, \\ & \qquad\qquad r^j \cdot (b - Ax) \geq 0, \quad \text{for all } j \in J, \\ & \qquad\qquad x \in \mathbb{Z}^n_+, \\ & \qquad\qquad \mu \in \mathbb{R} \end{aligned}$$

where $\{u^k \in \mathbb{R}^m_+ : k \in K\}$ and $\{r^j \in \mathbb{R}^m_+ : j \in J\}$ are the sets of extreme points and extreme rays, respectively, of $Q := \left\{u \in \mathbb{R}^m_+ | uG \geq h\right\}$, the feasible region of the dual of LP(x) [1].

However, the reformulation comes at a cost: While the number of complicating variables is reduced, the new problem usually contains a lot more constraints, possibly an exponential number of constraints, and exhaustively adding them requires the enumeration of all extreme points and extreme rays of the dual of LP(x). Instead, since most of these constraints will not be active in the optimal solution anyway, the sensible approach of the *Benders' decomposition* algorithm is as follows: We create a *master problem* by relaxing most (or all) of the constraints in the reformulated problem. We then solve the *master problem* and pass its solution to a subproblem which detects if (i) this solution is feasible for the original problem and (ii) (if feasible) if it is optimal for the original problem. If the solution is not feasible or feasible but not optimal, we obtain a violated constraint from the subproblem (along with an extreme point u^k or an extreme ray r^j) and add a corresponding cut to the *master problem*. We repeat this procedure until an optimal solution has been found. It should be noted that the effectivity of this approach strongly depends on the extent

[1] We denote by LP(x) the LP that we obtain from **(MIP)** by fixing x.

to which complicating variables can be (easily) separated among the constraints such that the original problem can be decomposed into multiple subproblems with a low degree of interdependence.

3.2 Suitability for this problem

As stated above, the main motivation of applying this algorithm to our problem is to reduce computation time. While the approach of *Benders' decomposition* is definitely not the only applicable method here, we believe that it is particularly suitable for our specific problem for mainly two reasons:

- Benders' decomposition provides an approach to separate the binary Unit Commitment problem from the continuous Economic Dispatch problem. It therefore seems like a fairly natural and intuitive method to solve the combined problem by splitting it into its original components, with the binary UC problem in the master problem, the subproblems solving the EC problem and an additional layer of sub-subproblems for the Security Constraints.

- The structure of the problem is such that it can be decomposed to a large degree, which - as explained above - is an important prerequisite for *Benders' decomposition* to be an efficient approach. On the one hand, the problem can be decomposed time-wise, such that each subproblem treats only a single hour rather than the entire day. This is possible, since once the Unit Commitment is fixed, Economic Dispatch for different hours can be planned independently from each other. On the other hand, the different contingencies can also be passed to different sub-subproblems in an additional layer, thereby further capitalizing on decomposition (and, possibly, parallelization).

3.3 Benders cut generation algorithm for the SC-UCED problem

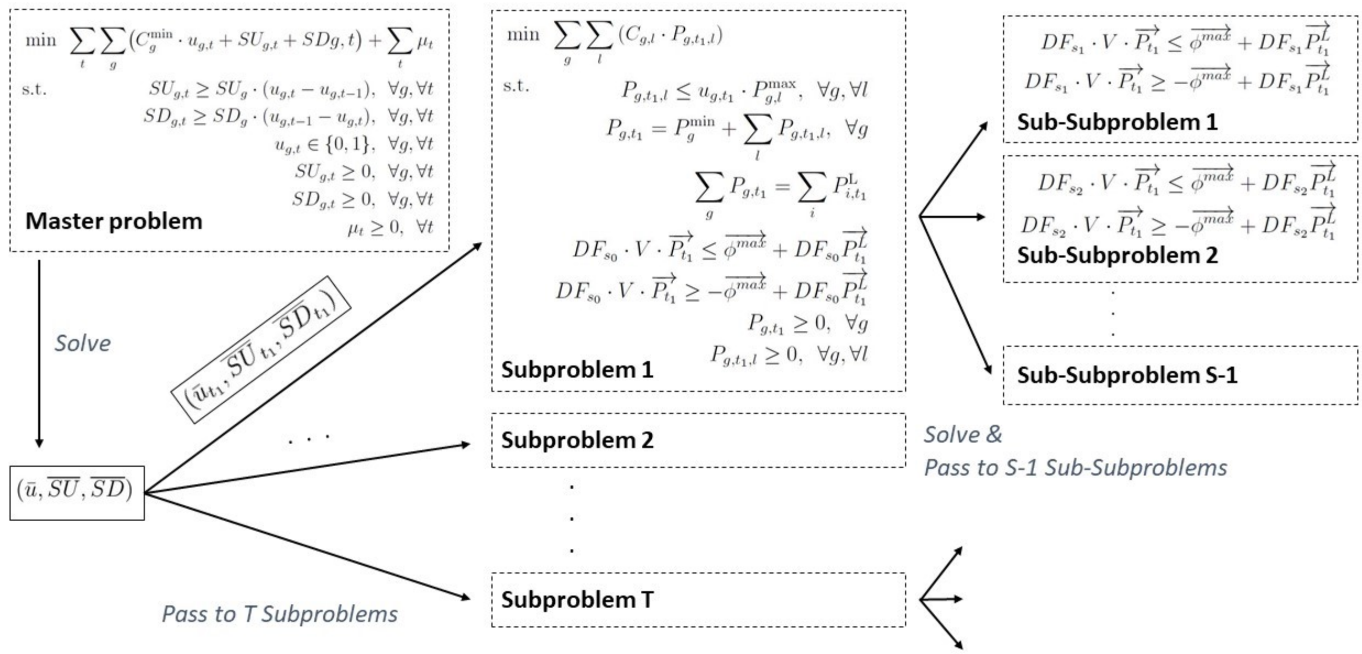
To solve the SC-UCED problem, we implement a cut generation algorithm based on *Benders' decomposition* with 3 layers (Algorithm 1, see Appendix for the Python implementation). The general layout of the decomposed problem in the algorithm as well as the cut generation is illustrated in Figure 2. The solution of the master problem - which includes only UC constraints - is passed to $|T|$ subproblems (the second layer), corresponding to the ED problem for different time periods $t = 1, \ldots, T$ for the base scenario. If applicable, each of the subproblems generates an optimality or feasibility cut that is then added to the master problem. Furthermore, in a third layer, each subproblem passes its individual optimal solution to $|S| - 1$ sub-subproblems (representing the $|S| - 1$ contingencies different from the base scenario). These sub-subproblems simply perform feasibility checks for the flow line constraints in the corresponding scenario and pass a feasibility cut to their associated subproblem, if any of these is violated. The master problem is solved repeatedly until either optimality is reached, or infeasibility is detected. Note that, in any case, the problem is bounded.

Algorithm 1: Cut generation algorithm

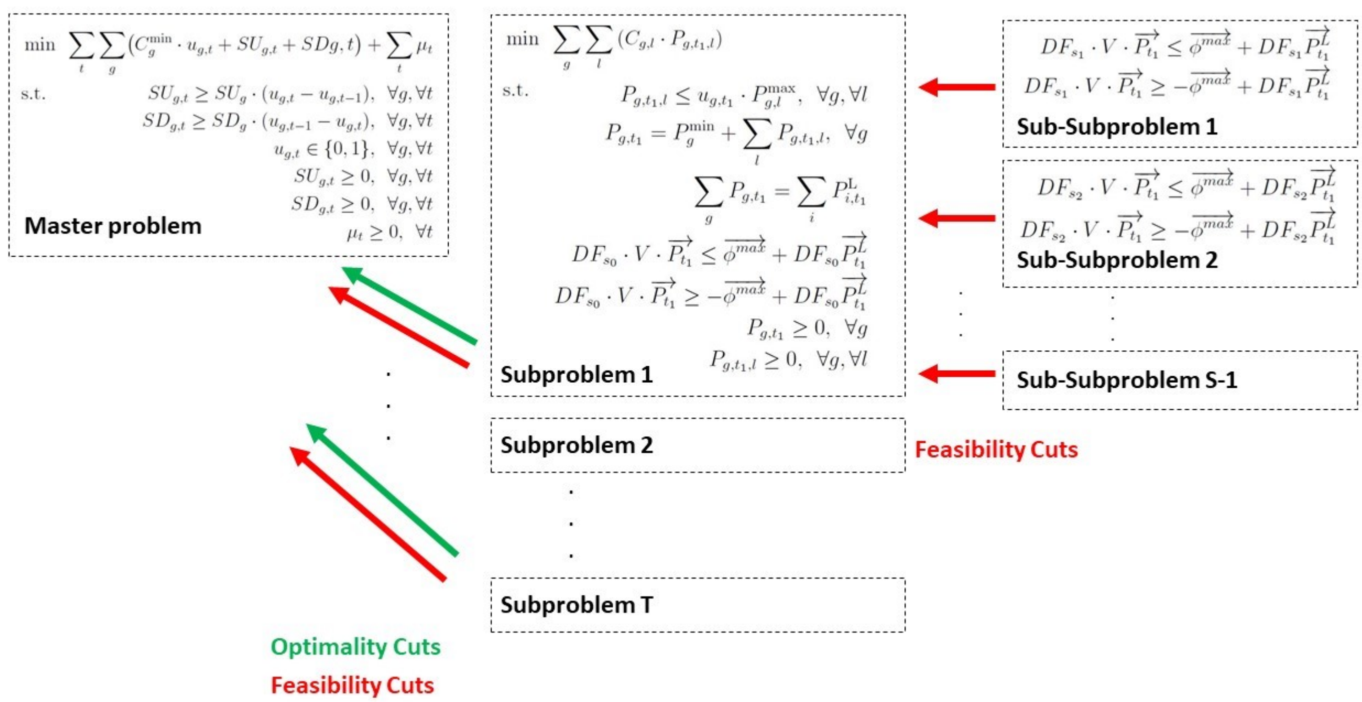
Input: SC-UCED parameters

Output: Optimal solution to the Master problem, if exists

1 Initialize Master problem: 24-hour UC with UC constraints only;
2 (1) *Solve Master problem* `// Cannot be unbounded due to problem formulation`;
3 **if** *Master problem is infeasible* **then** **stop** $\Rightarrow$ SC-UCED is infeasible ;
4 **else** pass the finite optimal solution $(\bar{u}, \overline{SU}, \overline{SD})$ to (2) ;
5 (2) *Solve Subproblems*;
6 Create subproblems $t = 1, \ldots, T$, i.e. the ED problem for each hour in the base scenario with UC variables fixed as $(\bar{u}_t, \overline{SU}_t, \overline{SD}_t)$;
7 **for** $t = 1, \ldots, T$ **do**
8   solve subproblem t : `// Cannot be unbounded due to problem formulation`;
9   **if** *infeasible* **then**
        `// the dual of subproblem` t `is unbounded (cannot be infeasible)`;
10   Add feasibility cut from subproblem t to Master problem;
11   **end**
12   **else**
13   obtain finite optimal solution $\bar{p}_t$;
14   **foreach** *contingency s not considered in subproblem t yet* **do**
15   check feasibility of $\bar{p}_t$ (i.e. line-flow constraints) ;
16   **if** *feasible for all contingencies s* **then**
17   Perform optimality check for $(\bar{u}_t, \bar{p}_t)$;
18   **if** *negative* **then** add optimality cut from subproblem t to Master Problem;
19   **end**
20   **else** add feasibility cuts for all violated s to subproblem t, re-solve subproblem t and go to line 9 ;
21   **end**
22 **end**
23 (3) *Master problem optimality*;
24 **if** *optimality checks were conducted and positive for all $t = 1, \ldots, T$* **then stop** $\rightarrow (\bar{u}, \bar{p})$ is optimal;
25 **else** return to (1);



(a) Decomposition into $|T|$ subproblems and $|T| \times |S-1|$ sub-subproblems



(b) Feasibility and optimality cuts

Figure 2: Cut generation procedure for the SC-UCED problem

4 Computations

4.1 Data and implementation

We used publicly available IEEE test systems of different sizes (30, 39, 118 and 145 nodes), with start-up and shutdown costs (SU_g, SD_g) randomly generated (within reasonable ranges). Demand was deterministically generated, using a reasonable arbitrary load profile curve to create data for 24 or 48 time periods. Computation time for the Benders' cut generation algorithm was compared against the benchmark, which uses Gurobi to solve **(MIP)** directly. For instance, in the example below the benchmark model was run for the IEEE 14 bus scenario for a simplified case without contingencies. Results can be seen in Figure 3. $SU_g = 0$ was used for all generators, and all the generators in this scenario have $P_g^{min} = 0$, which explains why generator 3 was committed even though it has 0 dispatch in the optimal solution.

```
Operating cost: 7683.47
Unit commitment solution
u[0,0] 1
u[1,0] 1
u[2,0] 1
u[3,0] 1
u[4,0] 1
Economic dispatch solution
pg[0,0] 200.789
pg[1,0] 42
pg[2,0] 0
pg[3,0] 10
pg[4,0] 6.21121
```

Figure 3: Screenshot of Gurobi implementation results for a single time period and single scenario.

The cut generation algorithm for the SC-UCED problem along with the code for data generation are included in the Appendix.

4.2 Results

The results obtained for different instances of the problem are listed in the table below. We make two main observations. First, since the objective function is (piecewise) linear, Benders' approach is a *reformulation*, meaning that it is an equivalent representation of the original problem. Hence, the results obtained must be the same for the benchmark and the approach proposed here. Indeed, for all instances tested, the same optimal solution was found when one was found, and infeasibility was attested by both methods in the single infeasible instance tried.

The second and most important result pertains to computation efficiency. The behavior that was expected was a poorer performance of the algorithm using *Benders' decomposition* compared to the benchmark for smaller instances, since the computational overhead caused by the creation of subproblems and sub-subproblems outweighs the savings of computational effort by the decomposition. For larger systems, however, *Benders decomposition*' is expected to perform better than a direct attempt to solve the complete **(MIP)** problem. In fact, this is indeed the behavior that we observe as systems grow in size, and especially as they grow in number of time periods. For the largest system tried, the Benders' approach solves a 48-period instance in 10 minutes, while the benchmark fails to in over 2 hours.

Nodes	Scenarios	Periods	Comp.time	Cuts	Benchmark CPU time	CPU time comparison
30	39	24	4.999	72	2.2715	120%
39	21	24	3.669	112	2.161	70%
39*	38*	24*	3.189*	96*	3.037*	5%*
39	21	48	5.996	213	2.807	114%
118	47	24	6.352	72	11.087	**-43%**
118	62	24	7.286	72	16.61	**-56%**
118	93	24	7.484	72	26.303	**-72%**
118	93	48	17.238	147	67.377	**-74%**
145	56	24	97.981	73	78.597	25%
145	75	24	166.131	74	133.064	25%
145	110	24	265.298	73	260.004	2%
145	110	48	658.829	151	>2h**	better than **-90%***

Results of algorithm implementation compared with benchmark (Gurobi)

$*$ Infeasible problem.

$**$ Presolve time : 627 s. ; process interrupted after 2h

5 Conclusions

We have successfully exploited the structure of the SC-UCED problem to apply *Benders' decomposition* in an algorithm that - according to our results - seems to significantly reduce the computation time for larger instances of the problem, compared to the benchmark.
Moreover, we have identified some implementation details with room for improvement that could further reduce computation time of the algorithm. These improvements relate to memory management and code flow in the optimization process. Specifically, for larger instances, time savings would probably be even more significant if taks were parallelized. These issues are on the coding side rather than on the side of optimization theory and hence not the focus of this paper.
Besides a larger range of computational experiments (possibly with more CPU power and more memory), future work may include more complicated generators (such as a combined cycle turbine or a pumped hydro storage unit) that require a set of additional logical constraints. These cases represent complicating constraints that couple different generating units by logical dependence (a steam turbine can only be used if its corresponding gas turbine is on) and inter-hour dependence (a pumped hydro storage unit can only be used as generator if it was filled earlier in the day). A sensible *Bender's decomposition* algorithm then needs to take into account this local interdependence of some constraints when decomposing the problem to the highest degree possible.

6 Appendix

6.1 Python Code for benchmark computation

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 15 17:47:18 2020

"""

from gurobipy import *
import numpy as np
import scipy.io as sio


#%% Get data from .mat file

baseMVA = 100 # MVA base for per-unit conversion
caseData = sio.loadmat('case145.mat')
pgl_max = np.array(caseData['pglmax'])
pg_min = np.array(caseData['pgmin'])
pl = np.array(caseData['PLT'])
cg = np.array(caseData['cg'])
phi_max = np.array(caseData['phi_max'])
DF = np.array(caseData['DF'])
V = np.array(caseData['V'])
SU = np.array(caseData['SU'])
SD = np.array(caseData['SD'])
DFs = np.array(caseData['DFs'])

#%% Initialize data arrays

ng = np.size(pg_min,0) # Number of generators
nseg = np.size(pgl_max,1) # Number of segments in linear approximation cost fcn
nsc = np.size(DFs,2) # Number of scenarios
nt = np.size(pl,1) # Number of periods in planning horizon
nb = np.size(pl,0) # Number of busses in the system

G = range(ng)
L = range(nseg)
S = range(nsc)
T = range(nt)
N = range(nb)

#%% Gurobi model

m = Model('DCOPF')

#============
# Decision Vars
#============
u = m.addVars(G,T, vtype=GRB.BINARY,name='u',
              lb=0,ub=1.0) # u_{g}
pg = m.addVars(G,T, vtype=GRB.CONTINUOUS,name='pg',
              lb=-GRB.INFINITY,ub=GRB.INFINITY) # p_{g,t}
pgl = m.addVars(G,T,L, vtype=GRB.CONTINUOUS,name='pgl',
              lb=0,ub=GRB.INFINITY) # p_{g,t,l}
sug = m.addVars(G,T, vtype=GRB.CONTINUOUS,name='sug',
```

```
                 lb=0,ub=GRB.INFINITY) # SU_{g,t}
sdg = m.addVars(G,T, vtype=GRB.CONTINUOUS,name='sdg',
                 lb=0,ub=GRB.INFINITY) # SD_{g,t}


#============
# Constraints
#============
# Coupling constraints
m.addConstrs((pgl[g,t,l]<=u[g,t]*pgl_max[g,l] for g in G for t in T for l in L))

# ED constraints
m.addConstrs((quicksum(pgl[g,t,l] for l in L)-baseMVA*pg[g,t]==-pg_min[g]
                                  for t in T for g in G),name='pcwLin') # pcwLin
m.addConstrs((quicksum(pg[g,t] for g in G)==quicksum(pl[i,t]
                            for i in N) for t in T),'power_bal') #power balance
for s in S:
    for t in T:
        DF = DFs[:,:,s]
        bV = np.reshape(DF@pl[:,t],[np.size(phi_max,0),1])
        m.addMConstrs(DF@V,pg.select('*',t),'<=',phi_max + bV,'flowLim+') # Transmission flow
        m.addMConstrs(-DF@V,pg.select('*',t),'<=',phi_max - bV,'flowLim-') # Transmission flow

# UC constraints
m.addConstrs((sug[g,0]>=u[g,0]*SU[g] for g in G),'SU0') # startup at t=0
m.addConstrs((sug[g,t]>=(u[g,t]-u[g,t-1])*SU[g] for g in G for t in T[1:]),'SUT') # startup at t>0
m.addConstrs((sdg[g,t]>=(u[g,t-1]-u[g,t])*SD[g] for g in G for t in T[1:]),'SDT') # shutdown


#============
# Objective
#============
m.modelSense = GRB.MINIMIZE
m.setObjective(quicksum(quicksum(sdg[g,t] + sug[g,t]+ u[g,t]*cg[g,0] +
                                  quicksum(pgl[g,t,l]*cg[g,1+l] for l in L) for g in G)for t in T))


# Run optimization
m.update()
m.optimize()

#%% Print solution
print('Operating cost: %g' % m.getObjective().getValue())
#print('Unit commitment solution')
#for v in u.select():
#    print('%s %g' % (v.varName,v.x))
#print('Economic dispatch solution')
#for v in pg.select():
#    print('%s %g' % (v.varName,baseMVA*v.x))
```

6.2 Benders cut generation algorithm for the SC-UCED problem

```python
from gurobipy import *
import numpy as np
import scipy.io as sio

#dynamic class
class Expando(object):
    pass


# ### Master Problem

# In[22]:


class Master:
    '''
    Parameters are
    Pmax_gl : array GxL        pgl_max = np.array(caseData['pglmax'])
    Pmin_g : array G           pg_min = np.array(caseData['pgmin'])
    Cmin_g : array G
    C_gl : array GxL           cg = np.array(caseData['cg'])
    SU_g : array G             SU = np.array(caseData['SU'])
    SD_g : array G             SD = np.array(caseData['SD'])
    PL_it : array NxT          pl = np.array(caseData['PLT'])
    DF_s_bi : array SxBxN (eg S arrays of size BxN)   DF = np.array(caseData
    ['DF'])
    V_ig : array NxG           V = np.array(caseData['V'])
    Phimax_b : array B         phi_max = np.array(caseData['phi_max'])
    along with the sets G,N,B,T,L,S
    '''
    def __init__(self,Pmax_gl,Pmin_g,Cmin_g,C_gl,SU_g,SD_g,PL_it,DF_s_bi,V_ig
    ,Phimax_b):#,M,N,P,b,A,c,G,h):
        #sets
        self.G=range(Pmin_g.shape[0])
        self.N=range(PL_it.shape[0])
        self.B=range(Phimax_b.shape[0])
        self.T=range(PL_it.shape[1])
        self.L=range(C_gl.shape[1])
        self.S=range(DF_s_bi.shape[2])

        #variables and constraints
        self.variables = Expando()
        self.constraints = Expando()
        #Data
        self.data = Expando()
        self._load_data()
        self._init_algo_data()
        #Model
        self._build_model()

    #Load the Data and Parameters
    def _load_data(self):
        self._load_coupling_params()
        self._load_continuous_params()

        self.data.SU_g=SU_g
        self.data.SD_g=SD_g

```

```
    def _load_coupling_params(self):
        self.data.Pmax_gl=Pmax_gl
        self.data.Cmin_g=Cmin_g
    def _load_continuous_params(self):
        self.data.Pmin_g=Pmin_g
        self.data.PL_it=PL_it
        self.data.DF_s_bi=DF_s_bi
        self.data.V_ig=V_ig
        self.data.Phimax_b=Phimax_b
        self.data.C_gl=C_gl


    #Initialize Algorithm successive variables
    def _init_algo_data(self):
        #stores the successive cuts added to the master pb
        self.data.cutlist = []

        #current bounds for the optimal solution to the original problem
        self.data.ub = GRB.INFINITY #upperbound of the optimal solution to the original pb
        self.data.lb = -GRB.INFINITY #lowerbound of the optimal solution to the original pb

        #stores the sequence of bounds for the optimal solution to the original problem
        self.data.ubs = []
        self.data.lbs = []
        #stores the sequence of variables x,y,and SPu of the successive (RMP) and (SP)
        self.data.xs = []
        self.data.ys = []
        self.data.SPus = []


    """
    Build the model
    """
    #Variables
    def _set_variables(self):
        m = self.model
        self.variables.u_gt = m.addVars(self.G,self.T,vtype=GRB.BINARY, name='u_gt')
        self.variables.SD_gt = m.addVars(self.G,self.T,vtype=GRB.CONTINUOUS, name='SD_gt')
        self.variables.SU_gt = m.addVars(self.G,self.T,vtype=GRB.CONTINUOUS, name='SU_gt')
        self.variables.mu = m.addVars(self.T,name='mu') #this will correspond to cost vector ^T the (Pgtl)s for each t in T
        m.update()


    #Objective function
    def _set_objective(self):
        self.model.setObjective(
            quicksum(self.data.Cmin_g[g]*self.variables.u_gt[g,t]+self.variables.SU_gt[g,t]+self.variables.SD_gt[g,t] for g in self.G for t in self.T)
            +quicksum(self.variables.mu[t] for t in self.T) ,
            GRB.MINIMIZE)

```

```
    #Constraints
    def _set_constraints(self):

        u_gt=self.variables.u_gt
        SD_gt=self.variables.SD_gt
        SU_gt=self.variables.SU_gt
        SU_g = self.data.SU_g
        SD_g = self.data.SD_g

        # Pure UC constraints (independent of sub-problems)
        self.constraints.uc ={}
        self.constraints.uc['SU0'] = self.model.addConstrs((SU_gt[g,0]>=u_gt[g,0]*SU_g[g] for g in self.G),'SU0') # startup at t=0
        self.constraints.uc['SUT'] = self.model.addConstrs((SU_gt[g,t]>=(u_gt[g,t]-u_gt[g,t-1])*SU_g[g] for g in self.G for t in self.T[1:]),'SUT') # startup at t>0
        self.constraints.uc['SDT'] = self.model.addConstrs((SD_gt[g,t]>=(u_gt[g,t-1]-u_gt[g,t])*SD_g[g] for g in self.G for t in self.T[1:]),'SDT') # shutdown

        # Cuts that will be added by sub-problems
        self.constraints.cuts = {} #empty set of constraints for the initial RPM

    #Model
    def _build_model(self):
        self.model = Model()
        self._set_variables()
        self._set_objective()
        self._set_constraints()
        self.model.setParam(GRB.Param.OutputFlag,0)
        self.model.update()


    #Updates bounds on the optimal solution to the original problem
    def _update_bounds(self):
        #z_sub = self.submodel.model.ObjVal
        z_subs = [self.submodel[t].model.ObjVal for t in self.T]
        z_master = self.model.ObjVal
        #self.data.ub = z_master - self.variables.mu.x + z_sub
        self.data.ub = z_master + quicksum(- self.variables.mu[t].x + z_sub[t] for t in self.T)
        self.data.lb = self.model.ObjBound

        self.data.ubs.append(self.data.ub)
        self.data.lbs.append(self.data.lb)


    def optimize(self, simple_results=False):

        m = self.model

        # Initial solution
        cont = 1

        dontStop = True
        while dontStop:
            dontStop = False

```

```
            #==========
            # Solve master problem
            #==========
            m.update() # Update since we added cuts
            m.optimize() #from this we get an optimal solution mu bar, x bar
            if (m.status==GRB.INFEASIBLE) or (m.status==GRB.INF_OR_UNBD) :
                print('Problem infeasible!!!!!')
                return


            if not hasattr(self,'submodel'):
                # Initialize submodels
                self.submodel = {}
                for t in self.T:
                    self.submodel[t] = Subproblem(self,t) # Build an instance of the subproblem (SP) from the initial solution

            else:
                # Update values of variables u_gt in subproblem
                for t in self.T:
                    for g in self.G:
                        for l in self.L:
                            self.submodel[t].constraints.coupl[g,l].rhs = self.variables.u_gt[g,t].x*self.data.Pmax_gl[g,l]
                    self.submodel[t].model.update()
                m.update()
            m.optimize()

            for t in self.T:
                subp = self.submodel[t] # For brevitiy below
                # Solve subproblem
                subp.optimize()

                # Case 1: subproblem infeasible
                if subp.model.status==GRB.INFEASIBLE or subp.model.status==GRB.INF_OR_UNBD :
                    dontStop = True # We'll have to iterate once more
                    rBar = np.array(subp.model.FarkasDual) # Unbounded ray of the dual
                    rCoupling = np.reshape(rBar[0:len(self.G)*len(self.L)],[len(self.G),len(self.L)]) # Get components corresponding to coupling constraints
                    constrs = subp.model.getConstrs()
                    # Add cut r(b-Ax)<=0,
                    # We are skipping the first |G|*|L| values of rBar, since these are the coupling constraints
                    # , which have rhs b equal to zero but constrs.rhs nonzero
                    # because of the Ax part of (b-Ax)
                    m.addConstr(-quicksum(rBar[mi]*constrs[mi].rhs for mi in range(len(self.G)*len(self.L)+1,len(constrs)))
                                -quicksum(self.variables.u_gt[g,t]*self.data.Pmax_gl[g,l]*rCoupling[g,l] for g in self.G for l in self.L)<=0)
#                    print('%d\t%d\t%d' % (cont,t,subp.model.status))
                elif subp.model.status==GRB.OPTIMAL:
                # Case 2: subproblem solved to optimality
                    if self.variables.mu[t].x-subp.model.ObjVal<-1E-3: # Using arbitrary tolerance of 1E-3
                        # Optimality constraint violated, add cut
                        dontStop = True # We'll have to iterate once more
```

```python
                        uBar = np.array(subp.model.Pi) # Shadow prices (optimal solution of the dual)
                        uCoupling = np.reshape(uBar[0:len(self.G)*len(self.L)],[len(self.G),len(self.L)]) # Get components corresponding to coupling constraints
                        constrs = subp.model.getConstrs()
                        # Add cut u(b-Ax)<= mu
                        # We are skipping the first |G|*|L| values of rBar, since these are the coupling constraints
                        # , which have rhs b equal to zero but constrs.rhs nonzero
                        # because of the Ax part of (b-Ax)
                        m.addConstr(quicksum(uBar[mi]*constrs[mi].rhs for mi in range(len(self.G)*len(self.L)+1,len(constrs)))
                                +quicksum(self.variables.u_gt[g,t]*self.data.Pmax_gl[g,l]*uCoupling[g,l] for g in self.G for l in self.L)<=
                                self.variables.mu[t])
#                    print('%d\t%d\t%d\t%4.2f' % (cont,t,subp.model.status,self.variables.mu[t].x))
                cont = cont + 1




# ### Subproblem

# In[6]:


# Subproblem
class Subproblem:
    def __init__(self, RMP,t):
        #sets
        self.G=RMP.G
        self.N=RMP.N
        self.B=RMP.B
        self.T=RMP.T
        self.L=RMP.L
        self.S=RMP.S

        # List of contingency scenarios that will be added to the set of
        # constraints. Initially, only the base case is considered (s=0)
        self.contScenarios = []
        self.contScenarios.append(0)

        # Time period
        self.t = t

        # Base MVA
        self.baseMVA = 100

        #variables and constraints
        self.variables = Expando()
        self.constraints = Expando()
        #Data
        self.data = Expando()
        #RMP
        self.RMP = RMP
        #Model
        self._build_model()
```

```python

    def optimize(self):
        m = self.model
        P_gtl = self.variables.P_gtl
        Pmin_g = self.RMP.data.Pmin_g
        PL_it = self.RMP.data.PL_it
        DF_s_bi = self.RMP.data.DF_s_bi
        V_ig = self.RMP.data.V_ig
        Phimax_b = self.RMP.data.Phimax_b
        P_gt = self.variables.P_gt
        t = self.t

        start = True
        overload = False
        while (start or overload):
            start = False
            if overload:
                m.update()
                overload = False

            m.optimize()
            if (m.status==GRB.INFEASIBLE) or (m.status==GRB.INF_OR_UNBD):
                # If infeasible, get back to master problem
                # Otherwise, go on
                return

            for s in self.S[1:]:
                # For each contingency, check overloads of branches
                if s in self.contScenarios:
                    # No need to check scenarios that are already in the subproblem
                    # formulation, those are guaranteed to be fine
                    continue

                DF = DF_s_bi[:,:,s]
                bV = np.reshape(DF@PL_it[:,t],[np.size(Phimax_b,0),1])
                P_gt_val = np.array([P_gt[g].x for g in self.G])# value of P_gt from solution
                flowVector = DF@V_ig@P_gt_val-bV
                if np.any(abs(flowVector)>Phimax_b):
                    # If there is overload, add scenario to problem and re-solve
                        overload = True
                        self.contScenarios.append(s)
                        m.addMConstrs(DF@V_ig,P_gt.select('*'),'<=',Phimax_b + bV,'flowLim+') # Transmission flow
                        m.addMConstrs(-DF@V_ig,P_gt.select('*'),'<=',Phimax_b - bV,'flowLim-') # Transmission flow


    """
    Build Subproblem (SP)
    """
    def _set_variables(self):
        m = self.model

        self.variables.P_gt = m.addVars(self.G, vtype=GRB.CONTINUOUS,name='pg',
              lb=-GRB.INFINITY,ub=GRB.INFINITY) # p_{g,t}
        self.variables.P_gtl = m.addVars(self.G,self.L, vtype=GRB.CONTINUOUS,
```

```python
    name='pgl',
            lb=0,ub=GRB.INFINITY) # p_{g,t,l}
        m.update()

    def _set_objective(self):
        m = self.model

        C_gl = self.RMP.data.C_gl
        P_gtl = self.variables.P_gtl

        m.setObjective(quicksum(C_gl[g,l]*P_gtl[g,l] for g in self.G for l in
     self.L))

    def _set_constraints(self):

        m = self.model
        P_gtl = self.variables.P_gtl
        Pmin_g = self.RMP.data.Pmin_g
        PL_it = self.RMP.data.PL_it
        DF_s_bi = self.RMP.data.DF_s_bi
        V_ig = self.RMP.data.V_ig
        Phimax_b = self.RMP.data.Phimax_b
        P_gt = self.variables.P_gt
        Pmax_gl = self.RMP.data.Pmax_gl
        t = self.t

        # Coupling constraints
        # ====================
        # Important: the uc_var constraints must be added first
        # We are counting on this to retrieve the unbounded rays when adding
        # cuts
        # ====================
        self.constraints.coupl = m.addConstrs((P_gtl[g,l]<=self.RMP.variables
    .u_gt[g,t].x*Pmax_gl[g,l] for g in self.G for l in self.L),'uc_var')

        # ED constraints
        # ==============
        # Pg = sum of piecewise linear bits
        self.constraints.pcwLin = m.addConstrs((quicksum(P_gtl[g,l] for l in
    self.L)-self.baseMVA*P_gt[g]==-Pmin_g[g] for g in self.G),name='pcwLin') #
    pcwLin
        # Power balance
        self.constraints.powBal = m.addConstr(quicksum(P_gt[g] for g in self.
    G)==quicksum(PL_it[i,t]
                                    for i in self.N),'power_bal') #power
    balance

        # Flow constraints in all scenarios considered
        for s in self.contScenarios:
            DF = DF_s_bi[:,:,s]
            bV = np.reshape(DF@PL_it[:,t],[np.size(Phimax_b,0),1])
            m.addMConstrs(DF@V_ig,P_gt.select('*'),'<=',Phimax_b + bV,'
    flowLim+') # Transmission flow
            m.addMConstrs(-DF@V_ig,P_gt.select('*'),'<=',Phimax_b - bV,'
    flowLim-') # Transmission flow


    def _build_model(self):
        self.model = Model()
        self._set_variables()
```

```
        self._set_objective()
        self._set_constraints()
        self.model.setParam(GRB.Param.OutputFlag,0)
        self.model.setParam(GRB.Param.InfUnbdInfo,1)
        self.model.update()

    def update_fixed_vars(self, RMP=None):
        pass


# ### To run the algorithm




#m = Master(M,N,P,b,A,c,G,h)


baseMVA = 100 # MVA base for per-unit conversion
caseData = sio.loadmat('case145.mat')
Pmax_gl = np.array(caseData['pglmax'])
Pmin_g = np.array(caseData['pgmin'])
PL_it = np.array(caseData['PLT'])
cg = np.array(caseData['cg'])
Phimax_b = np.array(caseData['phi_max'])
#DF = np.array(caseData['DF'])
V_ig = np.array(caseData['V'])
SU_g = np.array(caseData['SU'])
SD_g = np.array(caseData['SD'])
DF_s_bi = np.array(caseData['DFs'])
C_gl = cg[:,1:]
Cmin_g = cg[:,0]

m=Master(Pmax_gl,Pmin_g,Cmin_g,C_gl,SU_g,SD_g,PL_it,DF_s_bi,V_ig,Phimax_b)
m.optimize()
#%%
if m.model.status ==GRB.OPTIMAL:
    totCost = m.model.ObjVal
#     for t in m.T:
#         totCost += m.submodel[t].model.ObjVal
    print('Cost Benders Solution %4.3f' % totCost)
```

References

[1] S. Cvijic and J. Xiong, "*Security constrained unit commitment and economic dispatch through Benders decomposition: A comparative study*", 2011 IEEE Power and Energy Society General Meeting, Detroit, MI, USA, 2011, pp. 1-8.

[2] R. D. Zimmerman, C. E. Murillo-Sanchez, and R. J. Thomas, "MATPOWER: Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education," Power Systems, IEEE Transactions on, vol. 26, no. 1, pp. 12–19, Feb. 2011.

[3] A.W. Wood, B.F. Wollenberg, "*Power generation, operation, and control*", Hoboken, New Jersey, Wiley-IEEE, 2013.

[4] Kim, H., Sohn, H. S., Bricker, D. L. (2011). GENERATION EXPANSION PLANNING USING BENDERS'DECOMPOSITION AND GENERALIZED NETWORKS. International Journal of Industrial Engineering, 18(1).

[5] Soares, Joao, et al. "Two-stage stochastic model using benders' decomposition for large-scale energy resource management in smart grids." IEEE Transactions on Industry Applications 53.6 (2017): 5905-5914.

[6] Wolsey, L. A., Nemhauser, G. L. (1999). Integer and combinatorial optimization (Vol. 55), pp. 337-341, pp. 412-417. John Wiley Sons.