The OperationsResearch fall-2024 problem

Summary

The goal of project is to minimize the revenue of a factory which is producing metal alloys and sends them to markets using containers.

Pyomo formulation

We begin by importing the Pyomo package and creating a model abstract object:

```
from pyomo.environ import *
infinity = float('inf')
model = AbstractModel()
```

The sets Ore, $Alloys$, $Metals$, $Factories$, $Depots$ and $Markets$ are declared abstractly using the `Set` component:

```
model.Ore = Set()
model.Alloys = Set()
model.Metals = Set()
model.Factories = Set()
model.Depots = Set()
model.Markets = Set()
```

Similarly, the model parameters are defined abstractly using the `Param` component:

```
model.min_buy_fac = Param(model.Factories,within=NonNegativeReals,
default=0.0)
model.max_buy_fac = Param(model.Factories,within=NonNegativeReals,
default=infinity)
model.contract_cost = Param(model.Factories,within= NonNegativeReals)
model.A_comb_min = Param(model.Metals, within=NonNegativeReals,
default=0.0)
model.A_comb_max = Param(model.Metals, within=NonNegativeReals,
default=infinity)
model.B_comb_min = Param(model.Metals, within=NonNegativeReals,
default=0.0)
model.B_comb_max = Param(model.Metals, within=NonNegativeReals,
default=infinity)
model.price_of_alloy_fac = Param(model.Factories, model.Alloys,
within=NonNegativeReals)
model.Max_ore = Param(model.Ore,within=NonNegativeReals)
model.Ore_cost = Param(model.Ore,within=NonNegativeReals)
model.Ore_combination = Param(model.Ore, model.Metals,
```

```
within=NonNegativeReals)
model.container_cap = Param(within= NonNegativeIntegers)
model.Container_min_to_be_sent_depot = Param(model.Factories,
model.Depots, within=NonNegativeIntegers)
model.Container_Max_to_be_sent_depot = Param(model.Factories,
model.Depots, within=NonNegativeIntegers)
model.Container_cost_to_be_sent_depot = Param(model.Factories,
model.Depots , within=NonNegativeReals)
model.depots_min_to_receive = Param(model.Depots,
within=NonNegativeIntegers)
model.depots_Max_to_receive = Param(model.Depots,
within=NonNegativeIntegers)
model.Container_min_to_be_sent_market = Param(model.Depots,
model.Markets, within= NonNegativeIntegers)
model.Container_Max_to_be_sent_market = Param(model.Depots,
model.Markets, within= NonNegativeIntegers)
model.Container_cost_to_be_sent_market =
Param(model.Depots ,model.Markets, within= NonNegativeReals)
model.Max_market_demand = Param(model.Markets,model.Alloys, within=
NonNegativeReals)
model.Market_price = Param(model.Markets , model.Alloys , within=
NonNegativeReals)
```

The `within` option here is used in these parameter declarations to define expected properties of the parameters. This information is used to perform error checks on the data that is used to initialize the parameter components.

The `Var` component is used to define the decision variables: the `binary` is `{0,1}` to be clear.

```
model.Z = Var(model.Ore,model.Alloys, within=NonNegativeReals)
model.F = Var(model.Ore,model.Alloys, within=NonNegativeReals)
model.A = Var(model.Ore,model.Alloys, within=NonNegativeReals)
model.C = Var(model.Ore,model.Alloys, within=NonNegativeReals)
model.U = Var(model.Alloys,within=NonNegativeReals)
model.t = Var(model.Alloys,model.Factories,model.Depots,
within=NonNegativeReals)
model.Extracted_ore = Var(model.Ore,within=NonNegativeReals) # S in
report
model.h = Var(model.Factories,within= Binary)
model.B = Var(model.Factories, model.Depots,
within=NonNegativeIntegers)
model.g = Var(model.Alloys, model.Depots, model.Markets,
within=NonNegativeReals)
model.G = Var(model.Depots, model.Markets, within=
NonNegativeIntegers)
model.l = Var(model.Markets, within= Binary)
```

Rule functions are used to define constraint expressions in the `Constraint` component: here we have rule for maximum extraction of Ore:

```
def Max_extracted_ore_rule(model,i):
    return model.Extracted_ore[i] <= model.Max_ore[i]
model.Max_extracted_ore_limit =
Constraint(model.Ore,rule=Max_extracted_ore_rule)
```

Rule for Alloy weight limit(alloy weight is sum of metals weights in it):

```
def Alloy_sum_rule(model,j):
    return model.U[j] == sum(model.Z[i,j] for i in model.Ore)+\
                         sum(model.C[i,j] for i in model.Ore)+\
                         sum(model.A[i,j] for i in model.Ore)+\
                         sum(model.F[i,j] for i in model.Ore)
model.Alloy_sum_limit = Constraint(model.Alloys,rule=Alloy_sum_rule)
```

Rule for Metals in alloys(should be less than (or equal to) extracted metals from Ore):

```
def Metal_sum_rule_Z(model,i):
    return sum(model.Z[i,j] for j in model.Alloys) <=
model.Extracted_ore[i]*model.Ore_combination[i,'Zinc']
model.Metal_sum_limit_Z = Constraint(model.Ore,rule=Metal_sum_rule_Z)

def Metal_sum_rule_F(model,i):
    return sum(model.F[i,j] for j in model.Alloys) <=
model.Extracted_ore[i]*model.Ore_combination[i,'Iron']
model.Metal_sum_limit_F = Constraint(model.Ore,rule=Metal_sum_rule_F)

def Metal_sum_rule_C(model,i):
    return sum(model.C[i,j] for j in model.Alloys) <=
model.Extracted_ore[i]*model.Ore_combination[i,'Copper']
model.Metal_sum_limit_C = Constraint(model.Ore,rule=Metal_sum_rule_C)

def Metal_sum_rule_A(model,i):
    return sum(model.A[i,j] for j in model.Alloys) <=
model.Extracted_ore[i]*model.Ore_combination[i,'Aluminum']
model.Metal_sum_limit_A = Constraint(model.Ore,rule=Metal_sum_rule_A)
```

Rule for limitation of percentage of Metals in Alloys(f is bottom limit and t is top limit):

```
def Metal_in_alloy_rule_A_Z_f(model):
    value = sum(model.Z[i,'A'] for i in model.Ore)
    return model.A_comb_min['Zinc']*model.U['A']<=value
model.Metal_in_alloy_limit_A_Z_f =
Constraint(rule=Metal_in_alloy_rule_A_Z_f)
def Metal_in_alloy_rule_A_Z_t(model):
    value = sum(model.Z[i,'A'] for i in model.Ore)
    return value<=model.A_comb_max['Zinc']*model.U['A']
model.Metal_in_alloy_limit_A_Z_t =
Constraint(rule=Metal_in_alloy_rule_A_Z_t)
```

```
def Metal_in_alloy_rule_A_C_f(model):
    value = sum(model.C[i,'A'] for i in model.Ore)
    return model.A_comb_min['Copper']*model.U['A']<=value
model.Metal_in_alloy_limit_A_C_f =
Constraint(rule=Metal_in_alloy_rule_A_C_f)
def Metal_in_alloy_rule_A_C_t(model):
    value = sum(model.C[i,'A'] for i in model.Ore)
    return value<=model.A_comb_max['Copper']*model.U['A']
model.Metal_in_alloy_limit_A_C_t =
Constraint(rule=Metal_in_alloy_rule_A_C_t)

def Metal_in_alloy_rule_A_A_f(model):
    value = sum(model.A[i,'A'] for i in model.Ore)
    return model.A_comb_min['Aluminum']*model.U['A']<=value
model.Metal_in_alloy_limit_A_A_f =
Constraint(rule=Metal_in_alloy_rule_A_A_f)
def Metal_in_alloy_rule_A_A_t(model):
    value = sum(model.A[i,'A'] for i in model.Ore)
    return value<=model.A_comb_max['Aluminum']*model.U['A']
model.Metal_in_alloy_limit_A_A_t =
Constraint(rule=Metal_in_alloy_rule_A_A_t)

def Metal_in_alloy_rule_A_F_f(model):
    value = sum(model.F[i,'A'] for i in model.Ore)
    return model.A_comb_min['Iron']*model.U['A']<=value
model.Metal_in_alloy_limit_A_F_f =
Constraint(rule=Metal_in_alloy_rule_A_F_f)
def Metal_in_alloy_rule_A_F_t(model):
    value = sum(model.F[i,'A'] for i in model.Ore)
    return value<=model.A_comb_max['Iron']*model.U['A']
model.Metal_in_alloy_limit_A_F_t =
Constraint(rule=Metal_in_alloy_rule_A_F_t)

def Metal_in_alloy_rule_B_Z_f(model):
    value = sum(model.Z[i,'B'] for i in model.Ore)
    return model.B_comb_min['Zinc']*model.U['B']<=value
model.Metal_in_alloy_limit_B_Z_f =
Constraint(rule=Metal_in_alloy_rule_B_Z_f)
def Metal_in_alloy_rule_B_Z_t(model):
    value = sum(model.Z[i,'B'] for i in model.Ore)
    return value<=model.B_comb_max['Zinc']*model.U['B']
model.Metal_in_alloy_limit_B_Z_t =
Constraint(rule=Metal_in_alloy_rule_B_Z_t)

def Metal_in_alloy_rule_B_C_f(model):
    value = sum(model.C[i,'B'] for i in model.Ore)
    return model.B_comb_min['Copper']*model.U['B']<=value
model.Metal_in_alloy_limit_B_C_f =
Constraint(rule=Metal_in_alloy_rule_B_C_f)
def Metal_in_alloy_rule_B_C_t(model):
```

```
    value = sum(model.C[i,'B'] for i in model.Ore)
    return value<=model.B_comb_max['Copper']*model.U['B']
model.Metal_in_alloy_limit_B_C_t =
Constraint(rule=Metal_in_alloy_rule_B_C_t)

def Metal_in_alloy_rule_B_A_f(model):
    value = sum(model.A[i,'B'] for i in model.Ore)
    return model.B_comb_min['Aluminum']*model.U['B']<=value
model.Metal_in_alloy_limit_B_A_f =
Constraint(rule=Metal_in_alloy_rule_B_A_f)
def Metal_in_alloy_rule_B_A_t(model):
    value = sum(model.A[i,'B'] for i in model.Ore)
    return value<=model.B_comb_max['Aluminum']*model.U['B']
model.Metal_in_alloy_limit_B_A_t =
Constraint(rule=Metal_in_alloy_rule_B_A_t)

def Metal_in_alloy_rule_B_F_f(model):
    value = sum(model.F[i,'B'] for i in model.Ore)
    return model.B_comb_min['Iron']*model.U['B']<=value
model.Metal_in_alloy_limit_B_F_f =
Constraint(rule=Metal_in_alloy_rule_B_F_f)
def Metal_in_alloy_rule_B_F_t(model):
    value = sum(model.F[i,'B'] for i in model.Ore)
    return value<=model.B_comb_max['Iron']*model.U['B']
model.Metal_in_alloy_limit_B_F_t =
Constraint(rule=Metal_in_alloy_rule_B_F_t)
```

Rule for amount of exported alloy from main Factory, it should be less than(or equal to):

```
def Export_from_main_fac_rule(model,i):
    return model.U[i] >= sum(model.t[i,'Main',k] for k in
model.Depots)
model.Export_from_main_fac_limit =
Constraint(model.Alloys,rule=Export_from_main_fac_rule)
```

Rule of Limits of buying from factories:

```
def buy_from_fac_rule_f(model,i):
    value = sum(sum(model.t[j,i,k] for k in model.Depots)\
                                                for j in model.Alloys)
    return model.min_buy_fac[i]*model.h[i]<=value
model.buy_from_fac_limit_f= Constraint([1,2],rule=buy_from_fac_rule_f)
def buy_from_fac_rule_t(model,i):
    value = sum(sum(model.t[j,i,k] for k in model.Depots)\
                                                for j in model.Alloys)
    return value<=model.max_buy_fac[i]*model.h[i]
model.buy_from_fac_limit_t= Constraint([1,2],rule=buy_from_fac_rule_t)
```

Rule of limit for Alloys in one container from Factory to Depot:

```
def container_rule(model,i,j):
    return sum(model.t[a,i,j] for a in model.Alloys) <=
model.B[i,j]*model.container_cap
model.container_limit = Constraint(model.Factories, model.Depots,
rule=container_rule)
```

Rule of limit for transporting from fac to depots No1.:

```
def transportation_rule_t(model,i,j):
    return model.B[i,j]<=
model.Container_Max_to_be_sent_depot[i,j]*model.h[i]
model.transportation_limit_t =
Constraint(model.Factories,model.Depots, rule= transportation_rule_t)
def transportation_rule_f(model,i,j):
    return
model.Container_min_to_be_sent_depot[i,j]*model.h[i]<=model.B[i,j]
model.transportation_limit_f =
Constraint(model.Factories,model.Depots, rule= transportation_rule_f)
```

Rule of limit for transporting from fac to depots No2.:

```
def transportation_rule2(model,j):
    return inequality(model.depots_min_to_receive[j],sum(model.B[i,j]
for i in model.Factories),\
                      model.depots_Max_to_receive[j])
model.transportation_limit2 = Constraint(model.Depots,rule=
transportation_rule2)
```

Rule of limit for transporting from depots to markets.:

```
def transp_from_dep_to_marker_rule(model,i,k):
    return sum(model.t[i,j,k] for j in model.Factories) >=
sum(model.g[i,k,l] for l in model.Markets)
model.transp_from_dep_to_marker_limit =
Constraint(model.Alloys,model.Depots,\
                                                rule=
transp_from_dep_to_marker_rule)
```

Rule of limits for Alloys in containers transporting from depots to markets:

```
def container_rule2(model,i,j):
    return sum(model.g[l,i,j] for l in model.Alloys) <=
model.G[i,j]*model.container_cap
model.container_limit2 = Constraint(model.Depots, model.Markets,
rule=container_rule2)
```

Limit for containers to be sent to markets:

```
def market_sell_rule_f(model,i,j):
    return
model.Container_min_to_be_sent_market[i,j]*model.l[j]<=model.G[i,j]
model.market_sell_limit_f = Constraint(model.Depots,model.Markets,
rule= market_sell_rule_f)
def market_sell_rule_t(model,i,j):
    return
model.G[i,j]<=model.Container_min_to_be_sent_market[i,j]*model.l[j]
model.market_sell_limit_t = Constraint(model.Depots,model.Markets,
rule= market_sell_rule_t)
```

Here we have maximum market demands rule:

```
def max_market_demand_rule(model,k,i):
    return sum(model.g[i,j,k] for j in model.Depots) <=
model.Max_market_demand[k,i]
model.max_market_demand_limit = Constraint(model.Markets,
model.Alloys, rule= max_market_demand_rule)
```

The `Objective` component is used to define the revenue objective. This component uses a rule function to construct the objective expression:

$sense=maximize$ means we want to maximize the revenue.

```
def revenue_rule(model):
    return sum(sum(model.Market_price[m,j]*sum(model.g[j,k,m] for k in
model.Depots) for j in model.Alloys) for m in model.Markets)-\
           sum(model.Extracted_ore[i]*model.Ore_cost[i] for i in
model.Ore)-\
           sum(sum(model.price_of_alloy_fac[u,j]*sum(model.t[j,u,k]
for k in model.Depots) for j in model.Alloys) for u in
model.Factories)-\
           sum(model.h[u]*model.contract_cost[u] for u in
model.Factories)-\

sum(sum(model.Container_cost_to_be_sent_depot[i,j]*model.B[i,j] for j
in model.Depots) for i in model.Factories)-\

sum(sum(model.G[i,j]*model.Container_cost_to_be_sent_market[i,j] for j
in model.Markets) for i in model.Depots)

model.revenue = Objective(rule=revenue_rule, sense=maximize)
```