SunFlow Tutorial Part II

Supply Network Modeling & Optimization

SunFlow is a Python module for modelling and optimizing supply chain networks. Its name is derived from **Su**pply **n**etwork **Flow**.

This tutorial illustrates how you model and evaluate your supply chain networks with SunFlow. Step-by-step it tells you how to model simple and complex logistics networks including custom tariffs, normal and compound capacity constraints up to recipe based production and manufacturing networks covering substitutional products along the value chain.

- simple and complex logistics networks
- custom tariffs (coming up in the next release)
- normal and compound capacities
- networks with production/manufacturing incl. the usage of recipes
- substitutional products and alternatives

Introduction

Get an introduction how to use **SunFlow in 4 steps**.

Now let's start. Assume you are a supply chain architect who has the task to figure out how a given setup would look like under a set of constraints. The **constraints** are:

- one sourcing location 'Src'
- final distribution go to 2 US customers c1 and c2 with demands of 200t and 300t
- you consider employing 3 warehouses as distribution centers called 'Eastern WH', 'NorthEast WH' and NewOrleans WH'
- all distribution centers can be delivered from the sourcing location and each of them delivers directly to the customers
- the customers can be delivered as well directly from the sourcing location
- the handling costs at the warehouse are assumed variable, i.e. the higher the quantity handled the higher the cost. The artificial rates (=variable costs) are:
    - Eastern WH: 20
    - NorthEast WH: 10
    - NewOrleans WH: 25
- the freight rates (artificial as well) given here:

```
          from                   to              cost
          Freight(Src,           EasternPort,    180 )
          Freight(Src,           NewOrleansPort, 230 )
          Freight(Src,           NorthEast,      206 )
          Freight(NewOrleansPort, Midwest,       100 )
          Freight(NewOrleansPort, Texas,          40 )
          Freight(EasternPort,    Midwest,       100 )
          Freight(EasternPort,    Texas,         120 )
          Freight(NorthEast,      Midwest,        70 )
          Freight(NorthEast,      Texas,         150 )
          Freight(Src,            Midwest,       250 )
          Freight(Src,            Texas,         280 )
```

- You should answer the questions
    - what are the minimal total costs?
    - how does the flow across the distribution centers look like?

Step 1

First build up the initial network which is shown below.

```
sun = InitSunflow()

# Step 0: Build the network

n = SupplyNet('Distribution USA 0')

# define the activity nodes ==> define the flow

src           = n.source()

eastern_wh    = n.distribution(src)
northEast_wh  = n.distribution(src)
neworleans_wh = n.distribution(src)

c1            = n.delivery(src, eastern_wh, northEast_wh, neworleans_wh) .demand(200)
c2            = n.delivery(src, eastern_wh, northEast_wh, neworleans_wh) .demand(300)

n.close()
n.createGraph().view()
```

Now in detail. First you initialize SunFlow by calling InitSunFlow():

```
sun = InitSunflow()
```

After initialization you can model one or more supply networks. Here you go for one model by using SupplyNet() which is created simply by

```
n = SupplyNet('Distribution USA 0')
```

assigning the name 'Distribution USA 0' to it and returning a variable n which is further used to reference the network. The source src is setup by

```
src = n.source()
```

Next you create the 3 distribution sites by

```
eastern_wh    = n.distribution(src)
northEast_wh  = n.distribution(src)
neworleans_wh = n.distribution(src)
```

The customers sites are modeled accordingly by

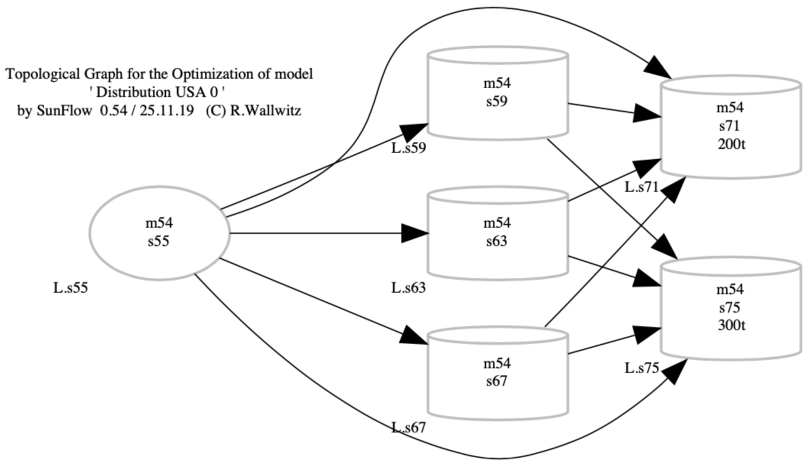
```
c1 = n.delivery(src, eastern_wh, northEast_wh, neworleans_wh) .demand(200)
c2 = n.delivery(src, eastern_wh, northEast_wh, neworleans_wh) .demand(300)
```

The parameters of delivery() reference the distribution centers and the source and indicate the potential sources of their receivals. To attach customer demands you use demand() with the corresponding quantity to be shipped to this customer. Your initial network is now complete for the final closure by

```
n.close()
```

To visualize your network you create a topological representation - a graph - and make it visible:

```
n.createGraph().view()
```



That's it. But it needs some refinement, so let's got step 2.

Step 2

Even though the network shows the basic links and structure you miss certainly a naming and the option to assign freight costs. The purpose of the 'mxx' labels and how you can avoid them will shown later (for the time being we just ignore them). Transports are always from a location to another or to itself. So you are going to enrich your network by locations which can you use later on for freight assignments. The modifications to be done are shown below:

```
Src            = Location('Manufacturer')
EasternPort    = Location('Eastern port')
NewOrleansPort = Location('NewOrleans port')
NorthEast      = Location('NorthEast')
Texas          = Location('Texas')
Midwest        = Location('Midwest')

Manufactur_WH = Producer('Manufactur')     .at(Src)
Eastern_WH    = Logistics('Eastern WH')    .at(EasternPort)
NorthEast_WH  = Logistics('NorthEast WH')  .at(NorthEast)
NewOrleans_WH = Logistics('NewOrleans WH').at(NewOrleansPort)
cust_Texas    = Customer('Cust Texas')     .at(Texas)
cust_Midwest  = Customer('Cust Midwest')   .at(Midwest)

n = SupplyNet('Distribution USA 2')

src           = n.source()            .at(Manufactur_WH)
eastern_wh    = n.distribution(src)  .at(Eastern_WH)
northEast_wh  = n.distribution(src)  .at(NorthEast_WH)
neworleans_wh = n.distribution(src)  .at(NewOrleans_WH)

c1 = n.delivery(src,eastern_wh,northEast_wh,neworleans_wh) .at(cust_Midwest).demand(200)
c2 = n.delivery(src,eastern_wh,northEast_wh,neworleans_wh) .at(cust_Texas)  .demand(300)
```

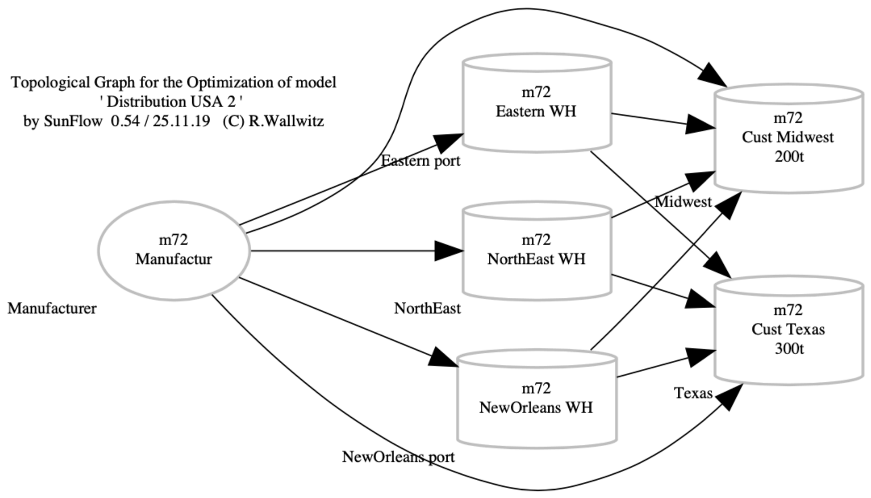
To define a location you use Location() with a name for this location:

```
EasternPort = Location('Eastern port')
```

Before go further you have specify the 'players' in your net as Producer, Logistics and Customer (there's more, but this will come later). Each player has attributes line a name, a location, a capacity and costs (fix & variable). You define a 'player' (actually it's a node in your network graph) by

```
Eastern_WH = Logistics('Eastern WH') .at(EasternPort)
```

saying Eastern_WH is a logistics hub called 'Eastern WH', is located at location EasternPort (by the at() statement) and is further referenced under Eastern_port. You apply this procedure to all of your nodes. Now the graph's structure is still the same, but it got now names and locations.



Step 3

In step 3 you assign freight and variable costs to the network.

Freights are attached by using the Freight() statement, below shown for a transport from Src to Eastern Warehouse. It requires a 'from' location (here Src), a 'to' location (EasternPort) and a freight rate (180):

```
        Freight(Src, EasternPort, 180 )
```

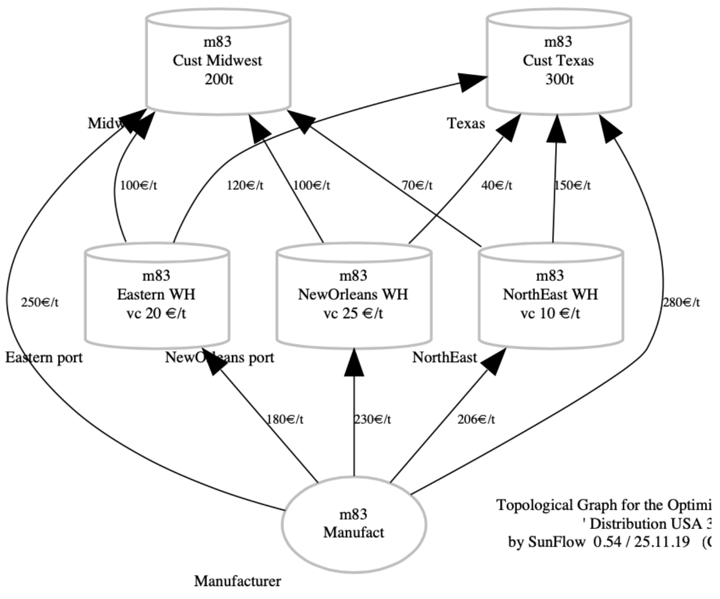
For all relations you want to consider in terms of cost by the model you have to assign rates in the same way. For those one not having any cost information their costs are assumed to be zero. For warehousing costs only variable costs are used (for simplicity here) which are assigned to the model as follows:

```
        eastern_wh    = n.distribution(BDE) .at(Eastern_WH)   .varcost(20)
        northEast_wh  = n.distribution(src) .at(NorthEast_WH) .varcost(10)
        neworleans_wh = n.distribution(BDE) .at(NewOrleans_WH).varcost(25)
```

For a better visualization you change the orientation of your model from left==>right (default) to bottom==>top by setting the createGraph() parameter orient to 'BT':

```
        n.createGraph(orient='BT').view()
```

turning your graph into this with variable costs ('vc') and freights (attached to arrows) now visible.



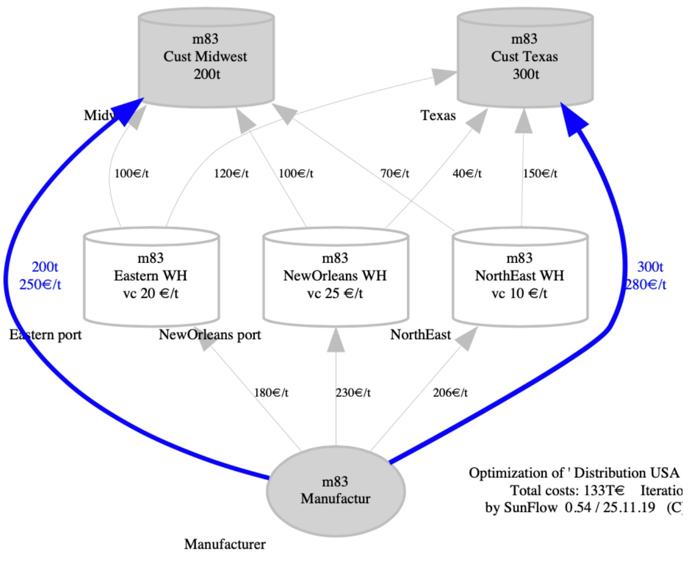
Step 4

Finally you need to let the flow running through your network to determine the optimal flow at lowest costs fulfilling all above mentioned constraints.

You do this simply by

```
        n.optimize().createGraph(orient='BT').view()
```

The statement optimize() optimizes your net, creates its graph and shows it.

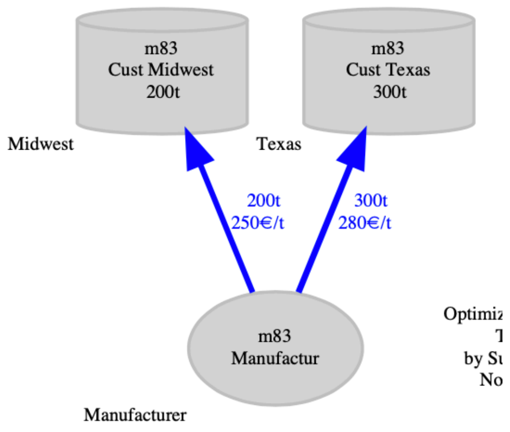


What does it say?
If you use the blue lines only, your network flow becomes cost optimal under the given constraints. The total costs are 133T currency units (default is €). The customer volumes are backward propagated across your network, here simply from c1 and c2 to your sourcing location.

Finally you may want to show only relations which are relevant within your optimized network. You can enforce SunFlow to do this by setting the flowOnly parameter to True,

```
        n.optimize().createGraph(flowOnly=True, orient='BT').view()
```

providing the final graph:



What does it mean?

It simply says that if you want to supply your two customers you should not use distribution centers. Instead ship it directly from your sourcing location to your customer if the lead-time allows it.

The final script or program looks as follows:

```
Src            = Location('Manufacturer')
EasternPort    = Location('Eastern port')
NewOrleansPort = Location('NewOrleans port')
NorthEast      = Location('NorthEast')
Texas          = Location('Texas')
Midwest        = Location('Midwest')

Freight(Src,            EasternPort,    180 )
Freight(Src,            NewOrleansPort, 230 )
Freight(Src,            NorthEast,      206 )
Freight(NewOrleansPort, Midwest,        100 )
Freight(NewOrleansPort, Texas,           40 )
Freight(EasternPort,    Midwest,        100 )
Freight(EasternPort,    Texas,          120 )
Freight(NorthEast,      Midwest,         70 )
Freight(NorthEast,      Texas,          150 )
Freight(Src,            Midwest,        250 )
Freight(Src,            Texas,          280 )

Manufactur_WH = Producer('Manufactur')      .at(Src)
Eastern_WH    = Logistics('Eastern WH')    .at(EasternPort)
NorthEast_WH  = Logistics('NorthEast WH')  .at(NorthEast)
NewOrleans_WH = Logistics('NewOrleans WH').at(NewOrleansPort)
cust_Texas    = Customer('Cust Texas')     .at(Texas)
cust_Midwest  = Customer('Cust Midwest')   .at(Midwest)


n = SupplyNet('Distribution USA 4')
src           = n.source()          .at(Manufactur_WH)
eastern_wh    = n.distribution(src) .at(Eastern_WH)     .varcost(20)
northEast_wh  = n.distribution(src) .at(NorthEast_WH)   .varcost(10)
neworleans_wh = n.distribution(src) .at(NewOrleans_WH)  .varcost(25)
c1 = n.delivery(src,eastern_wh,northEast_wh,neworleans_wh) .at(cust_Midwest).demand(200)
c2 = n.delivery(src,eastern_wh,northEast_wh,neworleans_wh) .at(cust_Texas)   .demand(300)
n.close()
n.createGraph(orient='BT').view()
n.optimize().createGraph(orient='BT').view()
n.optimize().createGraph(flowOnly=True, orient='BT').view()
```

Not too bad, isn't it?

Try to make it with Excel: it would more be more complicated, not readable for another person than you (just imagine the number of cells and their inter-related cryptic formulas) and hard to maintain and error-prone.

SunFlow provides an API that allows a straight forward definition of your network, is self-documenting, easy to understand and comes with powerful, comprehensive and customizable graphics capabilities and comes with a powerful optimization algorithm.

Even more, if you consider much more complicated and larger networks, covering compound capacities (!!!) and recipes (!!!), Excel can't handle it anymore due to the degree of complexity, but SunFlow can easily!

Working with Capacities

Capacity constraints are one of the typical and major constraints within a supply network. SunFlow handles two types of capacities:

- single capacities or just 'capacities' and
- compound capacities.

Single capacities belong to a single producer, supplier, distributor, ... defining their maximum capacity that can be applied. Compound capacities on the other hand belong to more than one producer, supplier, distributor, ... defining i.e. limiting the total of their joint capacities which can be applied.

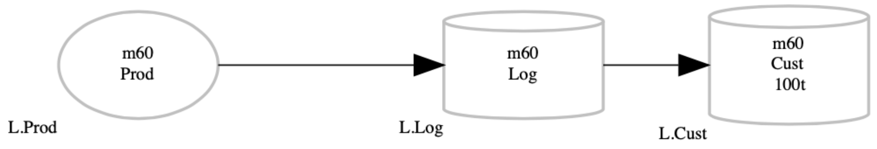
For instance, two manufacturers have one line each with a capacity of 100t each. But both are linked to a jointly used equipment - a reactor, which is assumed to be the bottleneck - that has a maximum capacity of 150t. Then the single capacities of both are totaling up to 200t, but the compound capacity just 150t limiting the parallel use of both lines to 150t. To what extent the single lines are used in parallel depends on other factors, primarily on the cost attributed to theses lines.

Single Capacities

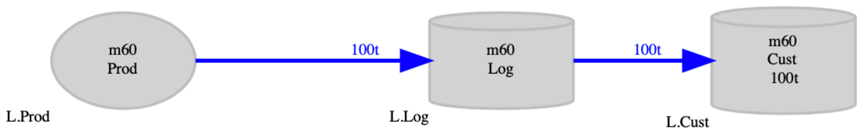
Assume a simple network consisting of a supplier (producer), a distributor (logistics) and a customer. The network would be setup like this,

```
producer = Producer('Prod')
log      = Logistics('Log')
customer = Customer('Cust')

n = SupplyNet('Capcacity 1')

prod = n.source()           .at(producer)
log  = n.distribution(prod).at(log)
cust = n.delivery(log)      .at(customer).demand(100)

n.close()

n.createGraph().view()
n.optimize().createGraph().view()
```

and will look like before optimization:



After optimizing the net, it looks like:



Everything is fine and the optimizer tells us just "successful". No capacities mentioned so far. Now turn on capacity inside the model by modifying the lines,

```
        prod = n.source()           .at(producer).capacity(200)
        log  = n.distribution(prod).at(log)      .capacity(300)
```

saying that the producer has a capacity of 200 and the distributor of 300 units (here t). With these capacities the customer's demand could be supplier at 100%. Instead of using capacity numbers in reality often capacity references are preferred like:

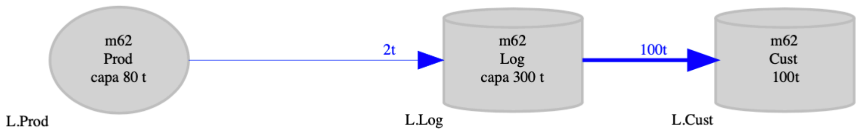
```
        capa_prod = Capacity(200)
        capa_log  = Capacity(300)

        prod = n.source()           .at(producer).capacity(capa_prod)
        log  = n.distribution(prod).at(log)      .capacity(capa_log)
```

capa_prod stands now for a capacity 200, capa_log for 300 and both are referenced at source() and distribution() statements. Whenever you change the Capacity statement, all references are updated immediately automatically to its new value.

Consider now a capacity constraint at the producer's site of 80t, i.e.

```
        capa_prod = Capacity(80)
        capa_log  = Capacity(300)
```

This would turn the optimized into

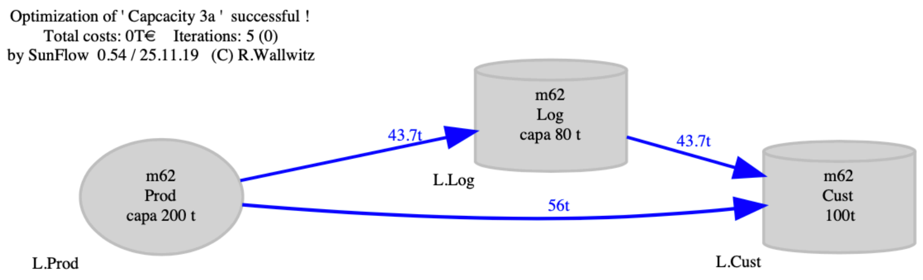


showing the capacities inside the boxes with for example "capa 300t" while signaling a "not successful" message. The numbers are attached at the arrows are not really meaningful in "not successful" cases, since the algorithm stops, if a condition that can't be satisfied at all.

To illustrate the capacity effect we switch the capacities among producer and distributor and add the option to supplier directly from producer to customer by modifying the delivery() statement

```
        cust = n.delivery(log,prod) .at(customer).demand(100)
```

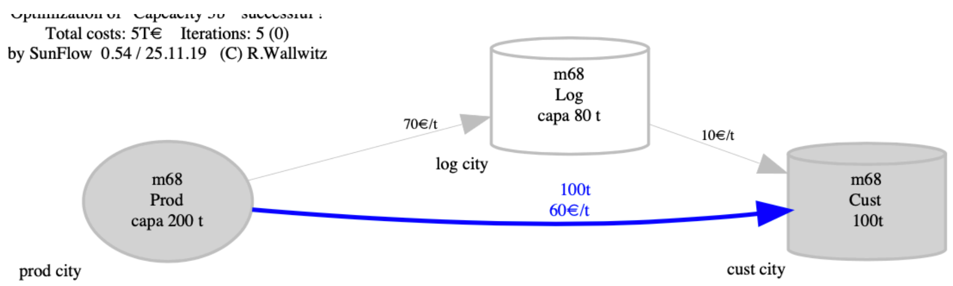
and see what happens:


Optimization of ' Capcacity 3a ' successful !
Total costs: 0T€ Iterations: 5 (0)
by SunFlow 0.54 / 25.11.19 (C) R.Wallwitz
m62
Prod
capa 200 t
L.Prod
43.7t
m62
Log
capa 80 t
L.Log
43.7t
56t
m62
Cust
100t
L.Cust

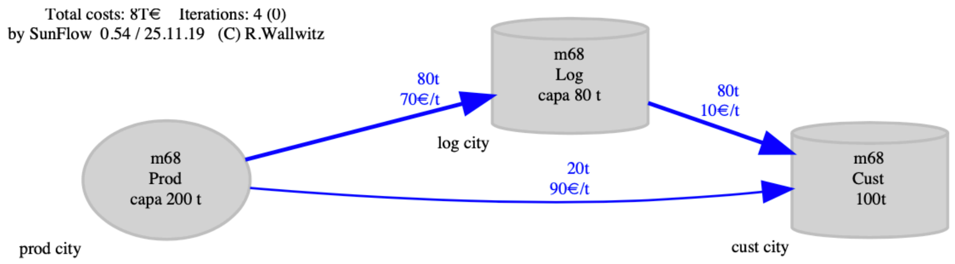
The optimizer indicates success and send now 44t (rounded up) through the distribution and 56t (rounded down) directly. If no cost - here freight - is given, an infinite number of allocations are possible. Let us change this now by defining freight to the links,

```
prod_loc = Location('prod city')
log_loc  = Location('log city')
cust_loc = Location('cust city')

producer = Producer('Prod').at(prod_loc)
log      = Logistics('Log').at(log_loc)
customer = Customer('Cust').at(cust_loc)

Freight(prod_loc,  log_loc,    70)
Freight(prod_loc,  cust_loc,   60)
Freight(log_loc,   cust_loc,   10)

capa_prod = Capacity(200)
capa_log  = Capacity(80)
```

turning the picture into



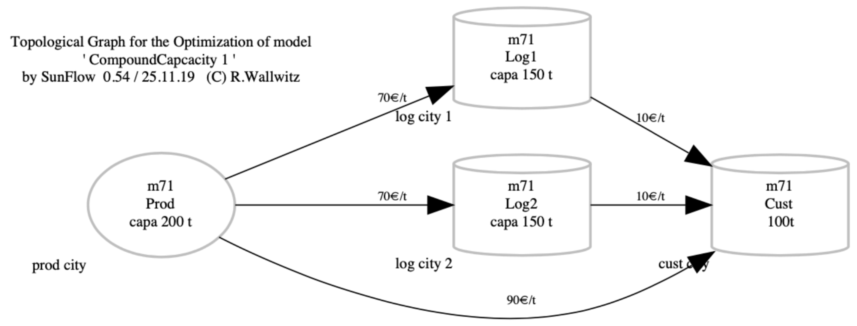
Why? If you add up the freight along the two ways, the path through distribution is 80€ (70+10), but the direct is just 60€. If we change the freight 'producer==>customer' from 60€ to 90€ it will become more expensive than the other one and the optimizer's answer will be:



Since the distributor path is the cheaper one, its maximum capacity of 80t is used and just the remaining volume of 20t is sent directly. The total cost is therefore 8.2T€ (=80*70+80*10+20*90) rounded to the shown 'Total costs: 8T€'.

Compound Capacities

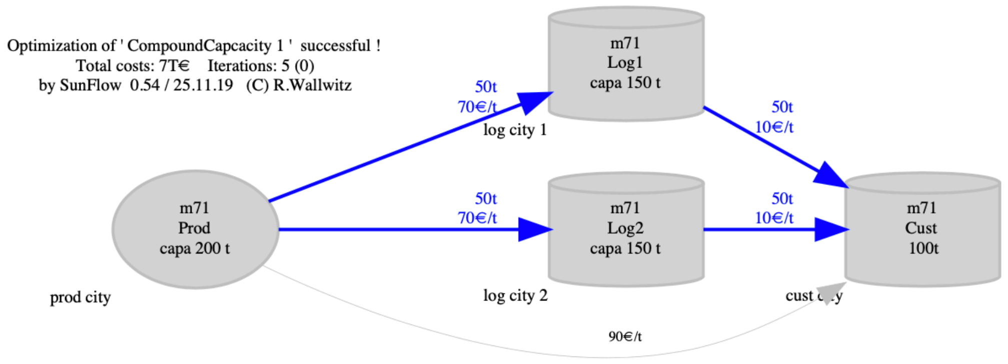
To illustrate the usage of compound capacities in SunFlow assume a network of a producer prod (the source at location 'prod city' with capacity of 200t), two distributors log1 and log2 (at locations 'log1 city' & 'log2 city' with individual capacities of 150t) and one customer (at location 'cust city' with a demand of 100t) like:



This net's code goes here:

```
prod_loc = Location('prod city')
log1_loc = Location('log city 1')
log2_loc = Location('log city 2')
cust_loc = Location('cust city')


producer = Producer('Prod') .at(prod_loc)
log1     = Logistics('Log1').at(log1_loc)
log2     = Logistics('Log2').at(log2_loc)
customer = Customer('Cust') .at(cust_loc)

Freight(prod_loc,  log1_loc,    70)
Freight(prod_loc,  log2_loc,    70)
Freight(prod_loc,  cust_loc,    90)
Freight(log1_loc,   cust_loc,   10)
Freight(log2_loc,   cust_loc,   10)

n = SupplyNet('CompoundCapcacity 1')

prod  = n.source()          .at(producer).capacity(200)
dist1 = n.distribution(prod).at(log1)     .capacity(150)
dist2 = n.distribution(prod).at(log2)     .capacity(150)

cust = n.delivery(dist1,dist2,prod) .at(customer) .demand(100)

n.close()
```

The optimizer calculates the cost optimized flow indicated by the blue arrows:



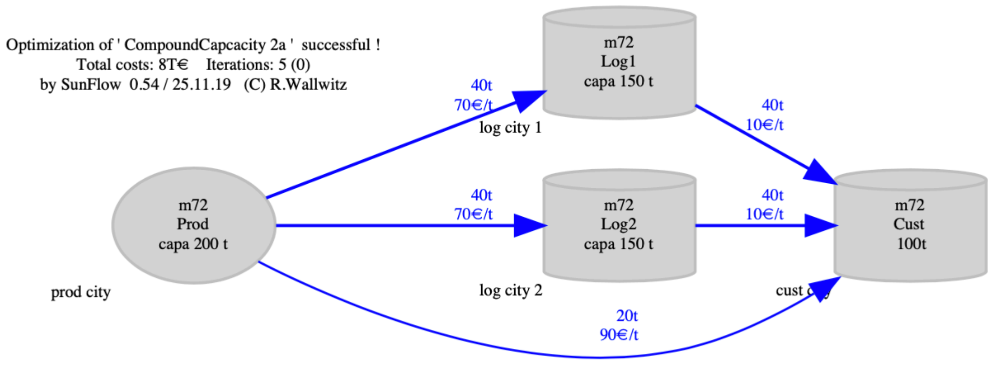
The flow goes through the distribution only, due to their freight cost advantage of 10€/t. Up to now there is no compound capacity yet. That's will be changed right now. Only 3 lines of code are changed:

```
capa_distrib  = Capacity('Distribution',80)

log1     = Logistics('Log1').at(log1_loc).compoundCapacity(capa_distrib)
log2     = Logistics('Log2').at(log2_loc).compoundCapacity(capa_distrib)
```

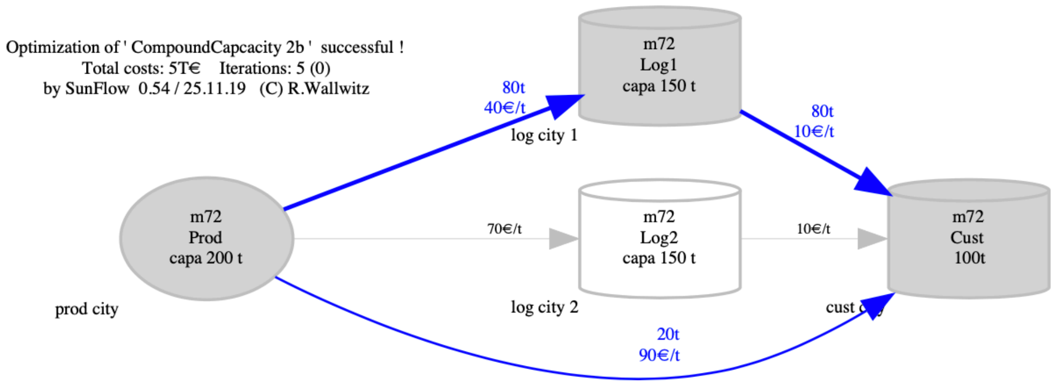
In the first line the capacity 'Distribution' is defined at 80t. The other 2 lines assign the distribution hubs log1 and log2 this capacity as a compound capacity by compoundCapacity(capa_distrib). The individual capacities still remain with 150t unchanged,

```
dist1 = n.distribution(prod).at(log1) .capacity(150)
dist2 = n.distribution(prod).at(log2) .capacity(150)
```

but their total simultaneously usable capacity is now limited to 80t. Let's see what the optimization algorithm will do with it:



The flow is split up even further. Still the path across the distribution is cheaper, but now limited to 80t due to a compound capacity of 80t. The remaining 20t flow directly from producer to customer. If the two ways through distribution have the same cost it typically allocates the flow equally among both. But if we reduce the freight from producer to log1 from 70/t down to 40€/t this picture will change to:



The distribution log2 is not used anymore due to the joint effects of cost and compound capacities:

- cheapest: prod ==> log1 ==> cust at 50€/t used
- runner-up: prod ==> log2 ==> cust at 80€/t not used
- most expensive: prod ==> cust at 90€/t used

Why is the runner-up with 80€/t not used? The joint capacity is already eaten up 100% by the path prod==>log1==>cust. So the path prod==>cust must be used to satisfy the customer demand even though it is the most expensive one.

Production & Manufacturing

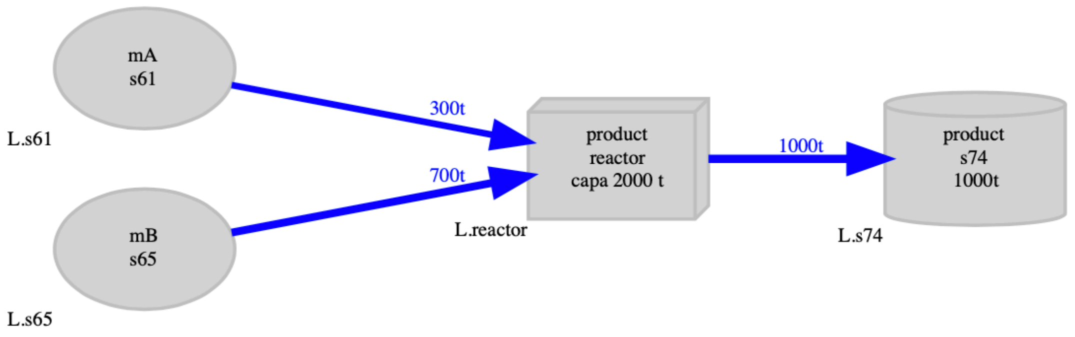
SunFlow does not make any difference between production and manufacturing. They are used synonymously.

Within a pure logistics network the products are not changed in their nature, they are just physically moved across the network. Costs are primarily driven by freight rates, warehouse and service charges as well as custom tariffs and taxation if the network crosses custom or tax boundaries. This must be enhanced if manufacturing is added to a network. SunFlow uses the term manufacturing for all steps where a product is created from one or more other products or components.

Recipes

To model such a step, a recipe is used. A recipe tells SunFlow what ingredients or components and how much of them are required to produce a product. The costs for a product are calculated from the values of their precursors - prices in case of raw materials or packaging, and the conversion cost (fix,var). Those costs are related to the manufacturing lines used, so they are not part of a recipe, but part of the manufacturing step and must be mentioned there.

For better understanding have a look on the following value chain:



The product named 'product' is made of 300t precusor 'mA' and 700t of precursor 'mB' in a reactor with a capacity of 2000t, where 1000t are consumed (by the customer) in the next value chain step. The model definition is:

```
mA = Material('mA')
mB = Material('mB')

product = Product('product').ingredient(mA,0.3).ingredient(mB,0.7)

reactor = Producer('reactor').capacity(2000)

n = SupplyNet('Recipe 1')

m_a = mA.raw(mA.name())
m_b = mB.raw(mB.name())

p_reactor = product.production( m_a, m_b).at(reactor)

n.delivery(p_reactor).demand(1000)

n.close()
```

Its recipe is

```
        product = Product('product').ingredient(mA,0.3).ingredient(mB,0.7)
```

and says that the product got two ingredients A and B. To produce one unit of product, 0.3 units of A and 0.7 units of B are required. You got to ensure that the sum of a recipe's weights is equal to 1.0. Otherwise your model would be wrong! You model a recipe with the Product() statement followed by ingredients(). A stepwise approach like

```
        product = Product('product')
        product = product.ingredient(mA,0.3)
        product = product.ingredient(mB,0.7)
```

would work as well.

The ingredients A and B are materials which must be created before used in a recipe. Here the materials are created by Material() and referenced by mA and mB respectively:

```
        mA = Material('mA')
        mB = Material('mB')
```

Within your net the most upstream components are modelled by the raw() statement,

```
        m_a = mA.raw(mA.name())
        m_b = mB.raw(mB.name())
```

referencing the materials mA and mB created before. To name them you refer to their names given at the time of creation by simply writing mA.name(). Alternatively you could have used as well

```
        m_a = mA.raw( 'A' )
        m_b = mB.raw( 'B' )
```

giving then the names 'A' and 'B' explicitely. The product itself is modeled by using the instance product

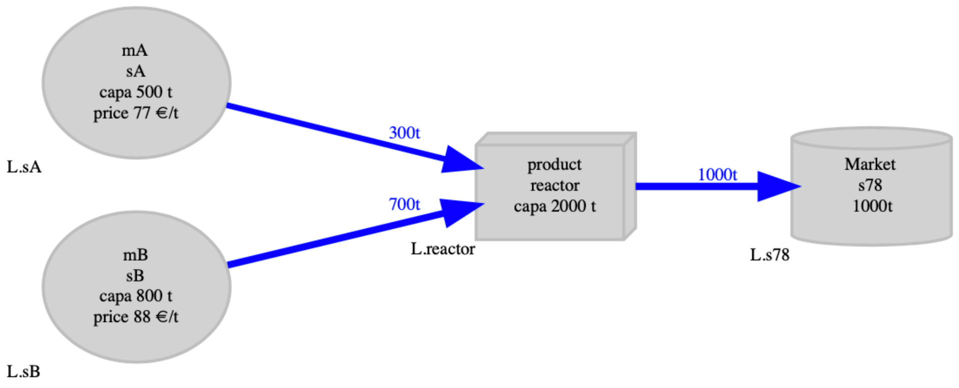
```
        p_reactor = product.production( m_a, m_b).at(reactor)
```

followed by production(m_a, m_b) where m_a and m_b represent the flow of raw materials A and B in your network.

Let's tune the model by adding further details. We're adding suppliers with their capacities and costs of raw materials - their prices. The changes are:

```
        sA      = Supplier('sA')      .capacity(500)
        sB      = Supplier('sB')      .capacity(800)
        reactor = Producer('reactor') .capacity(2000)

        m_a = mA.raw(mA.name()).by(sA).price(77)
        m_b = mB.raw(mB.name()).by(sB).price(88)
```

providing this model:



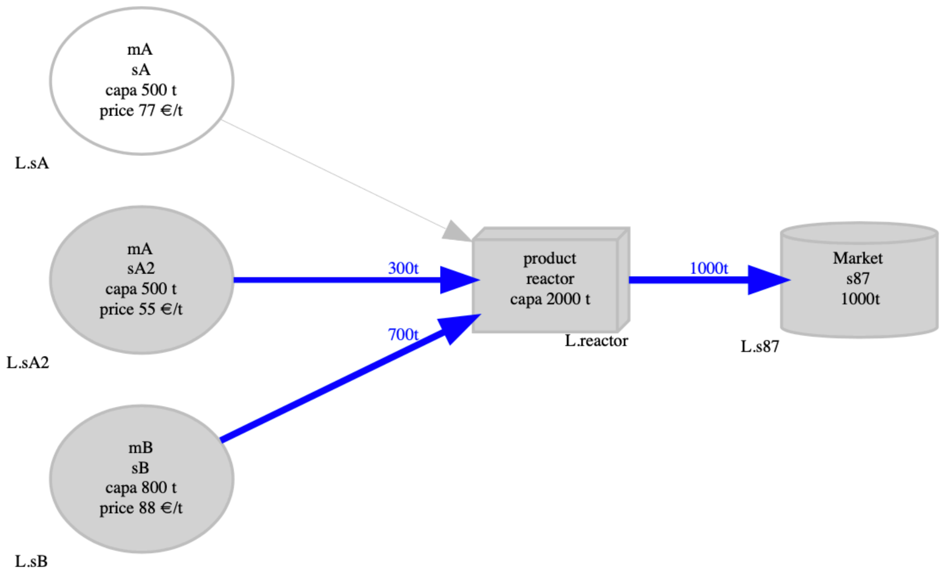
Alternative Raw Material Suppliers

In our model each raw material got one supplier. Now assume we have a second supplier for raw material A with a different price. Then two lines need to be added and one must be modified:

```
sA2  = Supplier('sA2').capacity(500)        # new
m_a2 = mA.raw(mA.name()).by(sA2).price(55) # new

p_reactor = product.production( m_a, m_a2, m_b).at(reactor) # modified
```

The two new lines represent the second supplier for A who is linked to raw material A simply with the statement 'by(sA2)'. You see, a material is by definition independent from a supplier, but needs to be linked to them. You should note as well the price of 55€/t which is lower than the one supplier sA with 77€/t.

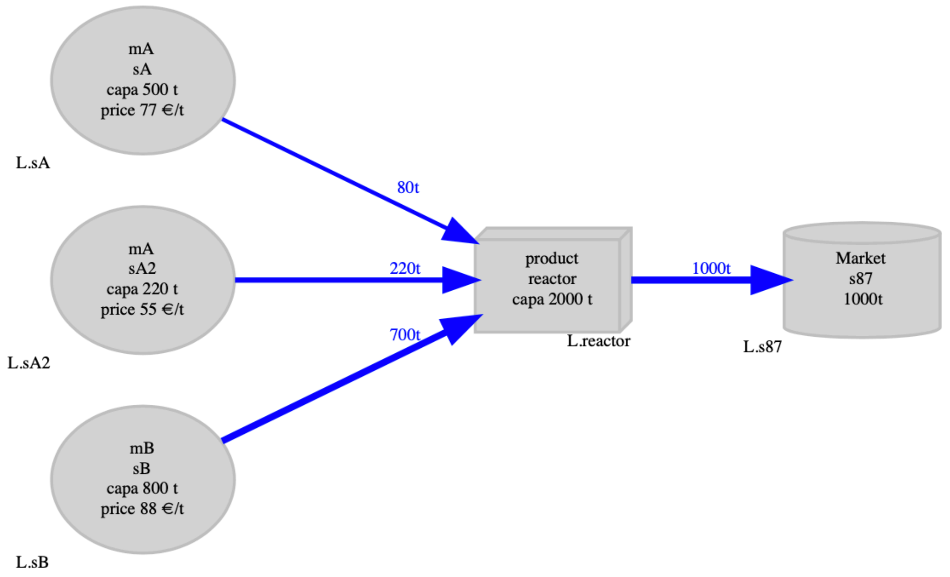
The production() statement was modified by adding the alternative raw material m_a2 into the arguments of production() telling SunFlow that it can use this raw material as a alternative for m_a from now on. With these modifications the optimized model becomes



clearly saying that you should prefer supplier sA2 instead of sA due to the lower price. This is a reasonable approach if suppliers has sufficient capacity to provide the requested volume of 300t. If you reduce supplier sA2's capacity down to 220t by

```
        sA2 = Supplier('sA2').capacity(220)
```

the situation changed to

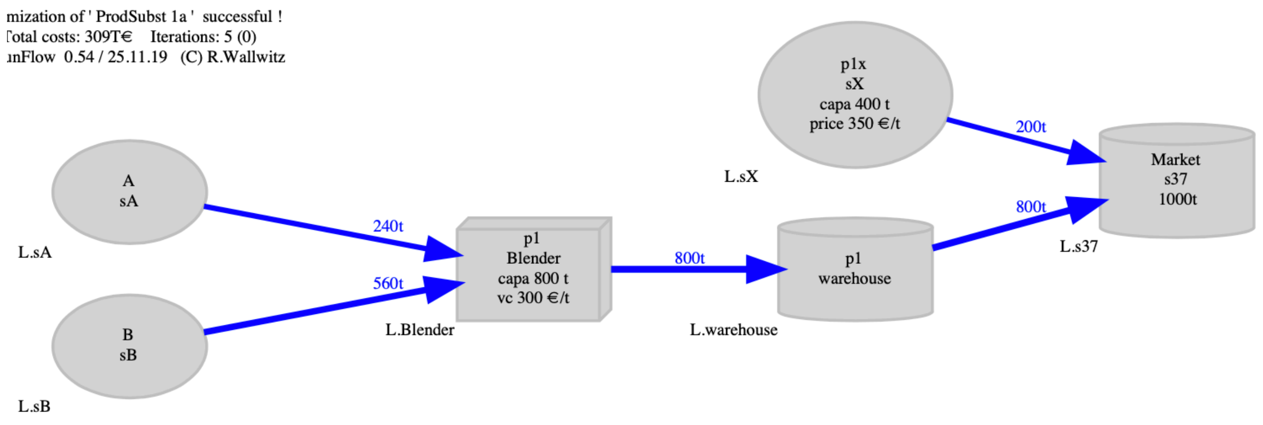


where the maximum available quantity of 220t is supplied by supplier sA2 and the remaining volume by supplier sA.

Substitution of Intermediate and final Products by alternative Suppliers

Within a supply network model you often has to answer the make-or-buy question. In the preceding section the modelling of the concept of alternative suppliers or raw materials was explained. But as well the manufacturing of intermediate and final products can be outsourced or manufactured on an alternative production line or plant. This section will show how those scenarios are modelled with SunFlow.

Here is the scenario we will look on.
The product p1 is produced from raw materials A and B at a blending facility (recipe identical the one in the previous section), shipped to and stored in a warehouse and finally transferred to the market. Alternatively there is an option to procure a substitute of p1, called p1x (x stands for external) at supplier sX and shipped directly to the market. The blender has a capacity of 800t and costs 300€/t while the 3rd party product p1X costs 350€/t for a maximum volume of 400t.



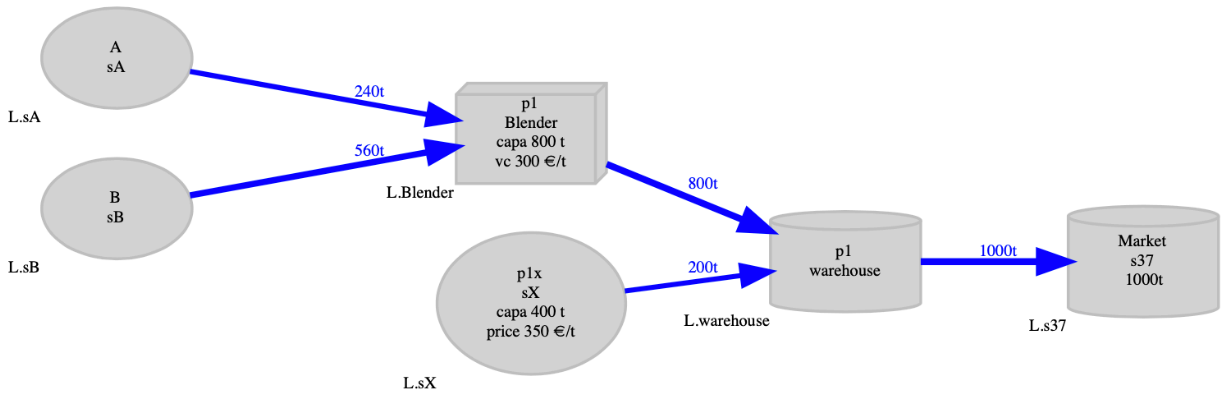
Since no other costs than conversion costs and product price for p1x is given, the 1000t are sourced in parallel via both paths. The model is:

```
mA   = Material('A')
mB   = Material('B')
p1   = Product('p1').ingredient(mA,0.3).ingredient(mB,0.7)
mP1x = Material('p1x').substituting(p1)

sA = Supplier('sA')
sB = Supplier('sB')
sX = Supplier('sX').capacity(400)

blender = Producer('Blender').capacity(800)
wh      = Logistics('warehouse')

n = SupplyNet('ProdSubst 1a')

m_a   = mA.raw(mA.name())     .by(sA)
m_b   = mB.raw(mB.name())     .by(sB)
m_p1x = mP1x.raw(mP1x.name()).by(sX).price(350)

p_blend = p1.production( m_a, m_b) .at(blender).varcost(300)
p_wh    = n.distribution(p_blend)  .at(wh)

n.distribution(p_wh,m_p1x).title('Market').demand(1000)
n.close()
```

The statement

```
mP1x = Material('p1x').substituting(p1)
```

tells SunFlow that p1x is identified as a substitute for p1. Both, product p1 and material p1x, are now treated by the algorithm as identical. This scenario assumes that p1x can only be supplied from sX directly to the market, even though it could be as well delivered first to the warehouse and then shipped to the market as shown here:

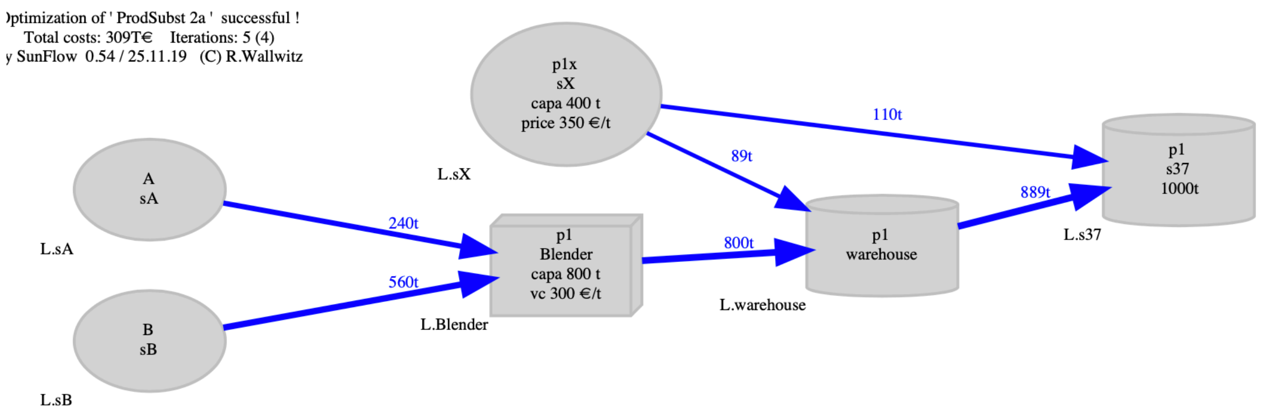


The only modification of the model is by changing two lines:

```
p_wh = n.distribution(p_blend, m_p1x).at(wh)
n.distribution(p_wh).title('Market').demand(1000)
```

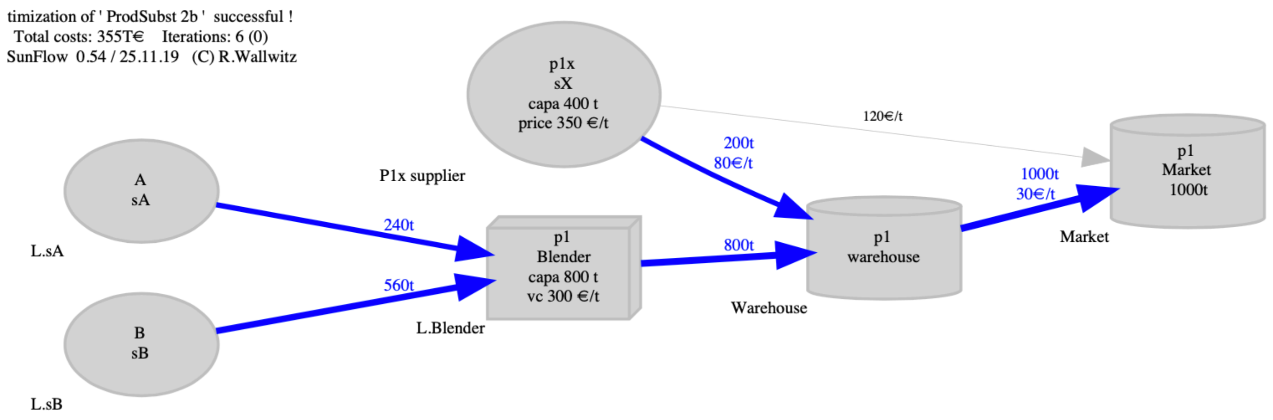
indicating that the flow of p1x goes now into the warehouse instead of going straight to the market.

But you actually don't want two models, you certainly would like to have one model covering both options letting the algorithm figure out the best of both ways. So put both models into one. Here it is:



)ptimization of ' ProdSubst 2a ' successful !
Total costs: 309T€ Iterations: 5 (4)
y SunFlow 0.54 / 25.11.19 (C) R.Wallwitz
p1x sX capa 400 t price 350 €/t
L.sX
A sA
L.sA
B sB
L.sB
p1 Blender capa 800 t vc 300 €/t
L.Blender
p1 warehouse
L.warehouse
p1 s37 1000t
L.s37
240t
560t
800t
89t
110t
889t

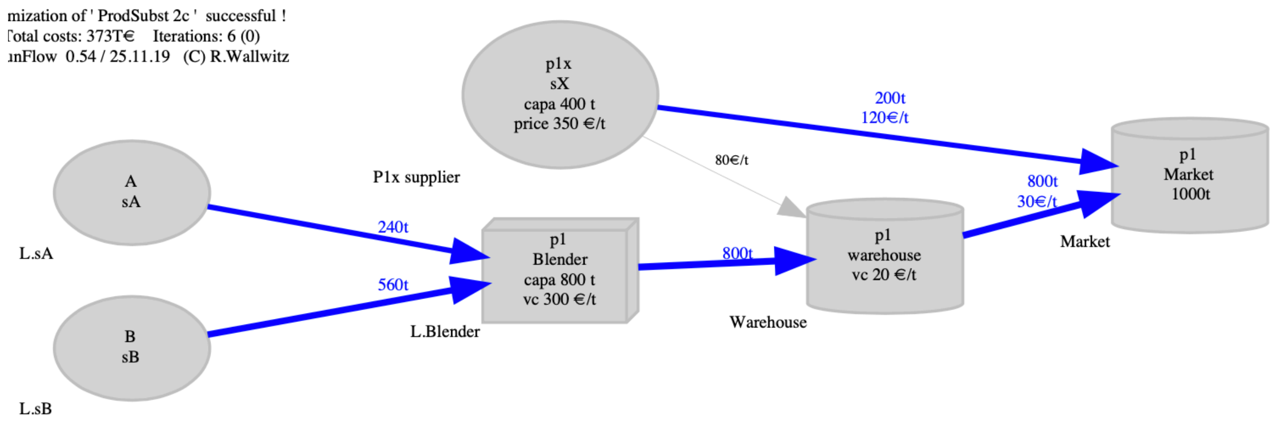
Now p1x has arrows pointing to the warehouse and the market simultaneously. The volumes following both pathways depending on the values of your model's parameters. We go now one step further and tuning the model by specifying the freights from p1x to market and warehouse and from warehouse to market which lead us to this model:



timization of ' ProdSubst 2b ' successful !
Total costs: 355T€ Iterations: 6 (0)
SunFlow 0.54 / 25.11.19 (C) R.Wallwitz
p1x sX capa 400 t price 350 €/t
P1x supplier
A sA
L.sA
B sB
L.sB
p1 Blender capa 800 t vc 300 €/t
L.Blender
p1 warehouse
Warehouse
p1 Market 1000t
Market
240t
560t
800t
200t 80€/t
120€/t
1000t 30€/t

Still the path through the blender is the cheapest - this why its capacity of 800t is fully utilized, but according to the freight as well for p1x the way through the warehouse is cheaper than direct shipment (120€/t > 80+30€/t).

As the final scenario assume now a service charge of 20€/t for the warehouse which is shown below.



mization of ' ProdSubst 2c ' successful !
Total costs: 373T€ Iterations: 6 (0)
ınFlow 0.54 / 25.11.19 (C) R.Wallwitz
p1x sX capa 400 t price 350 €/t
P1x supplier
A sA
L.sA
B sB
L.sB
p1 Blender capa 800 t vc 300 €/t
L.Blender
p1 warehouse vc 20 €/t
Warehouse
p1 Market 1000t
Market
240t
560t
800t
80€/t
200t 120€/t
800t 30€/t

Now the direct shipment of p1x from sX to the market becomes favorably. The related model is:

```
mA   = Material('A')
mB   = Material('B')
p1   = Product('p1').ingredient(mA,0.3).ingredient(mB,0.7)
mP1x = Material('p1x').substituting(p1)

sX_loc     = Location('P1x supplier')
wh_loc     = Location('Warehouse')
market_loc = Location('Market')

Freight(sX_loc, wh_loc,        80)
Freight(sX_loc, market_loc,   120)
Freight(wh_loc, market_loc,    30)

sA      = Supplier('sA')
sB      = Supplier('sB')
sX      = Supplier('sX')         .at(sX_loc) .capacity(400)
blender = Producer('Blender')                .capacity(800)
wh      = Logistics('warehouse').at(wh_loc)
market  = Customer('Market')     .at(market_loc)

n = SupplyNet('ProdSubst 2c')
m_a    = mA.raw(mA.name())      .by(sA)
m_b    = mB.raw(mB.name())      .by(sB)
m_p1x = mP1x.raw(mP1x.name()) .by(sX).price(350)

p_blend  = p1.production( m_a, m_b)      .at(blender)  .varcost(300)
p_wh     =  n.warehouse(p_blend, m_p1x) .at(wh)       .varcost(20)

n.delivery(p_wh,m_p1x).title('p1').demand(1000).at(market)
n.close()
```

Custom Tariffs & Taxes

Will be available soon.