SA405 · Advanced Math Programming · Fall 2020 · Goran, Lourenco, Skipper

Lesson 5

Python Function to Build Abstract Model

This lesson...

- We separate the model from the data by:
  - putting the model into a Python function
  - sending the data into the function as arguments to function parameters
- We use the BlackGold network flow problem as an example and see:
  - how to add upper bounds to indexed decision variables
  - how to index over a list of tuples (ARCS)
  - how to write a sum indexed over tuples where one of the values is fixed (to calculate "flow in" and "flow out" of a node)

Black Gold

This version of the max flow formulation does not use the dummy arc from sink to source.

Sets:

- $N =$ *the set of nodes (Optional. This set is never used in the model.*)
- $T =$ the set of transshipment nodes ($T \subseteq N$) (in this case, $T = \{1, 2, 3, 4\}$)
- $A =$ the set of directed arcs (pipes) (i, j), for some $i, j \in N$

Parameters:

- $c_{i,j} =$ the capacity of arc (i, j), for all $(i, j) \in A$

Decision Variables:

- $x_{i,j} =$ the amount of flow through arc (i, j), for all $(i, j) \in A$

Objective: Maximize the flow through the network (which is captured by finding the flow out of node 0)

Constraints:

- (1) Balance of flow through node n, for all $n \in T$
- (2) Enforce the capacity of arc (i, j), for all $(i, j) \in A$
- (2) Nonnegative* flow on all arcs $(i, j) \in A$

Maximize $F = \sum_{(i,j) \in A : i=0} x_{i,j}$

Subject to

- (1) $\sum_{(i,j) \in A : i=n} x_{i,j} - \sum_{(i,j) \in A : j=n} x_{i,j} = 0, \ \forall \ n \in T$
- (2) $x_{i,j} \leq c_{i,j}, \ \forall \ (i,j) \in A$
- (3) $x_{i,j} \geq 0, \ \forall \ (i,j) \in A$

* Note that we don't need to require integer flows due to the *max flow integrality theorem*.

Python function to build abstract model

- we send data (sets and parameter values) as arguments to the function
- this separates the model from the data
- we can put the model into a .py file, which we can `import` to use

Confusing Terminology

- Programming definitions:
  - a **parameter** is a variable in a function definition
  - an **argument** is the actual data that gets passed to the function
- LP / IP definition:
  - the **parameters** in an LP or IP are the constants, or numeric values, in the problem

Function parameters

- All data is passed into the function via parameters
- The IP or LP sets and **parameters** are the **arguments** that are passed into the **parameters** of the Python function that builds the model
- For the function parameters, I used the Python naming conventions for "variables"
  - one or two words (no more than 3!)
  - all lower case
  - words separated by _
- Whereas model parameters are constants, so I capitalized those
- (For consistency, you may wish to capitalize both model parameters and function parameters, since they end up representing the same objects.)

Return statement

- The function returns the model.
  - **The return statement is an easy line to forget and causes really strange looking errors.**

Commenting

- This example demonstrates the Python "docstring" to comment functions
- This allows anyone who uses the function to easily understand its behavior and expected parameters
- docstrings have very specific formatting conventions
  - Notice the triple quotes and lines/spacing
  - Information in the docstring:
    - the purpose of the function
    - description of expected arguments
    - what is returned by the function
  - See Python docstring documentation here (https://www.python.org/dev/peps/pep-0257/)

In [1]:
```
def max_flow(source, internal_nodes, arcs, capacity):
    """
    Build a max flow LP.

    Keyword arguments:
    source -- the source node
    internal_nodes -- all nodes except source and sink (list)
    arcs -- directed arcs in the netowork (list of tuples)
    capacity -- capacity of arcs (dictionary with arcs as keys)

    Return:
    Pyomo model
    """

    model = pyo.ConcreteModel()

    # add variables with bounds:  0 <= x[i,j] <= capacity[i,j]
    def bounds_rule(model,i,j):
        return (0,capacity[i,j])
    model.x = pyo.Var(arcs,
                      domain=pyo.NonNegativeReals,
                      bounds=bounds_rule)

    # add objective function (sum over flow out of the source node)
    def obj_rule(model):
        return sum(model.x[i,j] for i,j in arcs if i==source)
    model.obj = pyo.Objective(rule=obj_rule, sense=pyo.maximize)

    # add balance of flow constraints (indexed over transshipment nodes)
    def flow_balance_rule(model,node):
        return (sum(model.x[i,j] for (i,j) in arcs if i==node)
                == sum(model.x[i,j] for (i,j) in arcs if j==node))
    model.flow_balance_constraint = pyo.Constraint(internal_nodes,
                                                   rule=flow_balance_rule)

    # DON'T FORGET THE RETURN STATEMENT!!!
    return model
```

You can access docstring information about a function using `help(function_name)` :

In [2]:
```
help(max_flow)
```

```
Help on function max_flow in module __main__:

max_flow(source, internal_nodes, arcs, capacity)
    Build a max flow LP.

    Keyword arguments:
    SOURCE -- the source node
    INTERNAL_NODES -- all nodes except source and sink (list)
    ARCS -- directed arcs in the netowork (list of tuples)
    CAPACITY -- capacity of arcs (dictionary with ARCS as keys)

    Return:
    Pyomo model
```

New syntax in the network model above:

- The capacity constraints are incorporated as **upper bounds on decision variables**
  - we use the `bound_rule` helper function
  - `return (0,CAPACITY[i,j])` defines these bounds on `model.x[i,j]`:
    - lower bound: `0`
    - upper bound: `CAPACITY[i,j]`
- When we index over `ARCS`, we have to use `i,j` as index variables
  - (rather than a single index variable, like `arc`)
- `sum(model.x[i,j] for (i,j) in ARCS if i==node)`
  - sum over the arcs flowing out of the node called `node`
  - (i.e., the arcs that have `node` as the first index)

The rest of the code

Recompiling code

- Keep the `import` statement in a separate cell so you don't have to keep running it (it's slow)
- Recompile the model function whenever you change it (but otherwise it doesn't need to be recompiled)

In [3]:
```
# import Pyomo
import pyomo.environ as pyo
```

Passing data to the model function

- The parameters that are scoped to the code outside of the `max_flow` function are CAPITALIZED, for clarity
- Study the function call to build the model: `model = max_flow(...)`
  - The arguments are assigned to the function parameters via the names of the parameters in the function definition: `function_parameter=ARGUMENT_FROM_OUTER_CODE`
  - Order of arguments doesn't matter since we are using "Keyword Arguments", which we can specify by name rather than order.

Check for optimality & print

- Attempting to print a solution to a model that hasn't solved results in strange errors
- Check for "optimal" solver status before printing
- If status isn't "optimal", print status to help you understand what's wrong
  - Remember you can use `pyo.SolverFactory('glpk').solve(model, tee=True)` to see the output of the solver while it is running.

In [4]:
```
# Sets
TNODES = [1,2,3,4]
ARCS = [(0,1),(0,3),(1,2),(1,4),(2,5),(3,2),(3,4),(4,5)]

# Parameters
CAPACITY = {(0,1):9,(0,3):8,(1,2):5,(1,4):7,(2,5):10,(3,2):10,(3,4):7,(4,5):12}
SOURCE = 0  # This is the source node

# Call the function to build the model; pass model data as arguments
# (when we use function parameter names, order of arguments doesn't matter)
model = max_flow(internal_nodes=TNODES,
                 source=SOURCE,
                 arcs=ARCS,
                 capacity=CAPACITY)

# solve model
solver_result = pyo.SolverFactory('glpk').solve(model)

# Check if the model solved to optimality before printing solution
solve_status = solver_result.solver.termination_condition
if (solve_status=='optimal'):
    print(f'Max flow is {model.obj()}\n')
    for (i,j) in ARCS:
       print(f'The flow over {(i,j)} is {model.x[i,j].value}')
else:
    print(f'The solver status is {solve_status}')
```

```
Max flow is 17.0

The flow over (0, 1) is 9.0
The flow over (0, 3) is 8.0
The flow over (1, 2) is 5.0
The flow over (1, 4) is 4.0
The flow over (2, 5) is 10.0
The flow over (3, 2) is 5.0
The flow over (3, 4) is 3.0
The flow over (4, 5) is 7.0
```

IMPORTANT for debugging

- **Build your model cell-by-cell (as usual) before cutting and pasting to make the function**
- This makes debugging soooo much easier

Things to check

- Does your model function have a return statement?
- Are your function parameter names consistent in the function definition and in the function call?

In [ ]:
```

```