Project Report

George Frentzel

Linear Programming in Pyomo and other Python-based packages. An analysis on the feasibility of using Python-based software as a replacement for AMPL in ORIE 3300/3310

Foundation:

The current tool used for linear programming in ORIE 3300/3310 is AMPL, which comes with a few advantages and disadvantages. One of the main problems in using AMPL is that it is proprietary software that costs money. Due to this fact, students are unlikely to use it after graduation. Some benefits of using Python over a system like AMPL are listed below:

- Easy Implementation of Dynamic Programming
- Ability to take data from multiple sources
- Solver independence
- Ability to export data in convenient formats
- Ability to automate solving and re-solving problems
- Ability to reformulate a model and re-evaluate

Some downsides of using Python over AMPL are as follows

- Less documentation on Code
- Less clear error messages
- Less clarity in separation between models and data
- Less clarity in translation between LP notation and code

In investigating some options for software, I mainly investigated three Python packages. PuLP, GurobiPy, and Pyomo. In comparing them, I found Pyomo to be the most clear and easy to use, based on the main features of each package and how they compare in ease of coding. First, to understand the 3 implementations, listed below are 3 implementations of the linear program for maximum flow.

PuLP

```
from pulp import *
VERTICES = ['S',2,3,4,5,6,'T']
EDGES = [('S',2), ('S',3), (2,3), (2,4), (2,5), (3,5),
         (3,6), (4,'T'), (5,4), (5,6), (5,'T'),(6,'T')]

#Use a dictionary to associate capacities of each edge
capacity = {('S',2):1, ('S',3):7, (2,3):5, (2,4):6, (2,5):2,
(3,5):2, (3,6):4, (4,'T'):5, (5,4):1, (5,6):4, (5,'T'):4,
(6,'T'):3}

#Create variables for each edge
vars = LpVariable.dicts("Edges",EDGES,None,None,LpInteger)

#Sets capacity of each edge to be at least 0, at most its capacity
for a in EDGES: vars[a].bounds(0,capacity[a])

#Creates an LpProblem object, given as maximization and a title
prob = LpProblem("Max␣flow␣Problem", LpMaximize)

#Defines objective function
prob += lpSum([vars[(i,j)] for (i,j) in EDGES if j=='T'])

# Creates all problem constraints
for n in VERTICES:
if (n!='S' and n!='T'):
prob += (lpSum([vars[(i,j)] for (i,j) in EDGES if j == n]) ==
         lpSum([vars[(i,j)] for (i,j) in EDGES if i == n]))

prob.writeLP("maxflowlp.lp")
prob.solve()
print "Max␣flow␣=", value(prob.objective) for (i,j) in EDGES:
print (i,j), value(vars[(i,j)])
```

GurobiPy

```
from gurobipy import *
# Model data
VERTICES = ['S',2,3,4,5,6,'T']
EDGES = [('S',2), ('S',3), (2,3), (2,4), (2,5), (3,5),
         (3,6), (4,'T'), (5,4), (5,6), (5,'T'),(6,'T')]

#Use a dictionary to associate capacities of each edge
capacity = {('S',2):1, ('S',3):7, (2,3):5, (2,4):6, (2,5):2,
            (3,5):2, (3,6):4, (4,'T'):5, (5,4):1, (5,6):4,
            (5,'T'):4, (6,'T'):3}

# Create optimization model
m = Model('maxflow')

# create model variables
flowvar = {}
for i,j in EDGES:
    flowvar[i,j] = m.addVar()

#Need to update model after defining variables
m.update()
# define objective function
obfun = quicksum(flowvar[i,j] for i,j in EDGES.select(source,'*'))
m.setObjective(obfun, GRB.MAXIMIZE)

#add capacity constraints and flow conservations constraints
for i,j in EDGES:
    m.addConstr(flowvar[i,j]<=capacity[i,j])

for j in VERTICES:
    if j!=source and j!=sink:
        m.addConstr( quicksum(flowvar[i,j] for i,j
        in EDGES.select('*',j))
        == quicksum(flowvar[j,k] for j,k
        in EDGES.select(j,'*')))
# Compute optimal solution
m.optimize()
# Print solution
if m.status == GRB.Status.OPTIMAL:
    solution = m.getAttr('x', flowvar)
    print('\nOptimal␣flows:')
    for i,j in EDGES:
        if solution[i,j] > 0:
            print('%s␣->␣%s:␣%g' % (i, j, solution[i,j]))
```

Pyomo

```
from pyomo.environ import *
from pyomo.opt import SolverFactory
opt = SolverFactory('gurobi')

# Create the Abstract Model object
model = AbstractModel()
# Create the sets of VERTICES, edges and their relations
model.VERTICES = Set()
model.EDGES = Set(within=model.VERTICES * model.VERTICES)
model.source = Set(within=model.VERTICES)
model.sink = Set(within=model.VERTICES)
model.allothers = Set(within=model.VERTICES)
model.capacity = Param(model.EDGES)
model.flowvariable = Var(model.EDGES, within=NonNegativeReals)

# define the objective function of the problem
def NetFlow(model):
    sum( model.flowvariable[i, j]
        for (i, j) in model.EDGES
        if j in model.sink )

# incorporate into problem
model.maxFlow = Objective(rule=NetFlow, sense=maximize)

# define capacity constraint function and add capacity constraints
def CapacityConstraint(model, i,j):
    return ((model.flowvariable[i,j] <= model.capacity[i,j]))

model.loadOnArc = Constraint(model.EDGES, rule=CapacityConstraint)

# define flow constraints and add flow constraints to problem
def FlowConservation(model, i):
    amountIn = sum(model.flowvariable[j, i]
                   for (j, i) in model.EDGES)
    amountOut = sum(model.flowvariable[i, j]
                    for (i, j) in model.EDGES)
    return (amountIn == amountOut)

# Define flow constraints for new set allothers (all except S & T)
model.flow = Constraint((model.allothers), rule=FlowConservation)

instance = model.create_instance('MaxFlow.dat')
results = opt.solve(instance)
instance.display()
```

AMPL data file

```
set VERTICES := 1 2 3 4 5 6 7;
set EDGES := (1,2) (1,3) (2,3) (2,4) (2,5) (3,5)
             (3,6) (4,7) (5,4) (5,6) (5,7) (6,7);
set source := 1;
set sink := 7;
set allothers := 2 3 4 5 6;

param: capacity :=
1 2 1
1 3 7
2 3 5
2 4 6
2 5 2
3 5 2
3 6 4
4 7 5
5 4 1
5 6 4
5 7 4
6 7 3;
```

AMPL Model File

```
# maxflow model file
set VERTICES;
set EDGES within (VERTICES cross VERTICES);
param SOURCE in VERTICES;
param SINK in VERTICES;
param capacity {EDGES} >= 0;

var FLOW {(i,j) in EDGES} <= capacity[i, j], >= 0;

maximize NetFlow:
        sum {(i,j) in EDGES: j = SINK} FLOW[i,j]
        - sum {(i,j) in EDGES: i = SINK} FLOW[i,j];

subject to FlowConserv {i in VERTICES: i <> SOURCE and i <> SINK}:
        sum {(i,j) in EDGES} FLOW[i,j]
        - sum {(j,i) in EDGES} FLOW[j,i] = 0;
```

Max Flow Problem Implementations

First, we will look at the PuLP code for the max flow problem. The first step that needs to be done is to load the data in using python data structures, typically lists for sets, and dictionaries for parameters of sets. In this example, lists are used to define the vertices and edges, while a dictionary is used to associate a capacity with each edge. A key aspect of PuLP is that it is difficult to use preexisting AMPL data with the models, without having to convert them to python data. I tested a feature called AMPLy (a part of PuLP), which is designed to read partial AMPL data files to take in data, but found it very cumbersome to use, and wholly impractical. AMPLy only provides functions that parse lines of AMPL data code into python data structures, so requires the user to parse the lines of the .dat files, and input each line into the functions. In addition, it only supports some AMPL data structures, so does not work with all .dat files.

The next step in the code is to create LP variables from the set EDGES, and PuLP has the ability to require the values that the variables to take on integer values or continuous values, with the keywords "LpInteger" or "LpContinuous". Next, PuLP requires the user to manually input the bounds of each variable, and then create a "problem" object, with a name and specifying whether it is a maximization or minimization problem. The main problem with PuLP comes in the next part, when adding in the objective function and constraints to the "problem" object. There is no clear difference in adding the objective function and adding constraints to the problem. PuLP recognizes the first statement added with the += operator to be the objective function, provided the user inputs and expression rather than an equation. The rest of the statements added to the problem are recognized as constraints, provided an equation is added in. In a lot of the linear programs discussed in ORIE 3300 & ORIE 3310, some general constraints will apply to a whole set of variables or inputs. For example, in the maximum flow problem, a net flow constraint (where flow into a node must equal flow out of a node) applies to all nodes besides the source and the sink. The process of adding constraints like those requires the user to iterate with a for loop over the set, adding in the constraints one by one. Finally PuLP requires the linear program to be translated to a .lp file format, and then solved, rather than solved directly.

The second python package, GurobiPy, is a similar implementation to PuLP in many ways. Like PuLP, it cannot read AMPL data files, and all of the data must be loaded in using python data structures. Then, a model object is created, and the objective function and constraints are added in in a similar way, by iterating over the set of EDGES or VERTICES. GurobiPy distinguishes itself from PuLP by having a clearer separation of objective functions and constraints, as well as the ability to add multiple constraints at once. However, GurobiPy is solver dependent, and can only use the Gurobi solver.

The last example of code before the AMPL code is the linear program for maximum flow in Pyomo. Pyomo has the advantage of being the most like AMPL in much of its structure. Once an abstract model has been created, similar to AMPL, one must define all of the sets and parameters that are used in the model. Then, like PuLP and GurobiPy, one must add the objective functions and constraints to the model. This works differently in Pyomo in that the user writes functions, or rules for the objective function and constraints, and then applies it to the sets. For example, in the above code, a function NetFlow is defined, and then the objective function is applied to the model in a statement that defines the objective function following the Netflow rule. Similarly, for adding the constraints for the capacity, the first step is to define a function CapacityConstraint, and then apply the rule to the set EDGES. Once the objective function and constraints are added into the model, the model is instantiated with data from an AMPL data file, and then a solver is called on it. Pyomo distinguishes itself from PuLP and GurobiPy in that it can take data from AMPL data files, and has a clear separation between the model and data files.

Comparison

In the logic of implementing linear programs, PuLP and GurobiPy are very similar. The main difference in PuLP or GurobiPy versus Pyomo is the way that constraints are added. With PuLP/GurobiPy, the common method is to use for loops to iterate over sets, adding a constraint in each instance, while in Pyomo, a function or rule is defined, and applied to sets. While both work, the Pyomo implementation is a little more useful in that when repeating code, for example applying a certain constraint to two sets, it is much easier to simply apply the same function to another set, than write code out in a for loop again. Additionally, Pyomo is able to read AMPL data files, and has a structure quite similar to AMPL, and thus also to linear programming formulations as defined in ORIE 3310, in defining sets and parameters. Finally, in researching and testing out implementations of linear programs in each of these modules, while in general there is insufficient documentation for all three of these tools, Pyomo has the most complete documentation, which enabled ease of implementing features used in AMPL and ORIE 3310. Overall, Pyomo distinguishes itself as the most useful, and the best for use in teaching ORIE 3300 and 3310 by having many features that cant be found in PuLP or GurobiPy. Overall, to replace AMPL, Pyomo must have a certain set of features to be considered just as or more useful than AMPL in teaching ORIE 3300 & 3310. Some of the features are as follows

- Ability to model simple linear programs
- Ability to model integer programs
- Dynamic Programming
- Solve and then resolve models, adding constraints or variables

In addition to this, as Pyomo is implemented in a more general-purpose language, it has several features that AMPL does not have:

- Automation of re-solving models
- Easy Dynamic Programming Implementations
- Ability to implement Branch and Bound
- Interactions between different Linear Programs
- Ability to return values besides for display

Dynamic Programming

As seen from the previous example of code, Pyomo has the ability to model simple linear programs and integer programs. In addition to that, it also can solve dynamic programming problems. The method to solve dynamic programming problems in AMPL is very crude. Below shows the example code from recitation 5 of solving a problem of a knapsack problem, where the weight is the amount of pennies that we can spend, and value is the amount of wins that the team gets.

AMPL Model File

```
param N; # number of teams
param pennies;  # maximum amount we can spend
param seeds; # of different seeds
param team {n in 1..N} symbolic; # team name
param seed {n in 1..N}; # seed of team
param wins {n in 1..N}; # number of wins of team
param seedcost {n in 1..seeds} >=0 integer; # cost of a given seed

set options {n in 1..N, s in 0..pennies} :=
    # check if we have the option to take team n in a given state
    {i in 0..1: i*seedcost[seed[n]] <= s };

param f {n in 1..N+1, s in 0..pennies} :=
    # max wins from teams n..N using at most s pennies
   if n=N+1 then 0   # no more items left to consider
      else
         max {i in options[n,s]}
                (i*wins[n] + f[n+1,s-i*seedcost[seed[n]]]);

param DPvalue := f[1,pennies]; # compute the optimal value
set opt {n in 1..N, s in 0..pennies} :=
     # optimal decisions
  {i in options[n,s]:
            f[n,s]=i*wins[n]+ f[n+1,s-i*seedcost[seed[n]]]};
```

AMPL Data File

```
param N := 32;  #number of teams
param pennies := 100; #amount we can spend
param seeds := 16; #number of different seeds
param: team wins seed :=
1 'Michigan␣St'    4  1
2 'Fresno␣St'      1  9
3 Gonzaga          2  12
```

```
4 'Indiana␣St'      1  13
5 Temple            3  11
6 Florida           1  3
7 'Penn␣St'         2  7
8 'North␣Carolina'  1  2
9 Illinois          3  1
10 Charlotte        1  9
11 Syracuse         2  5
12 Kansas           2  4
13 'Notre␣Dame'     1  6
14 Mississippi      2  3
15 Butler           1  10
16 Arizona          5  2
17 Duke             6  1
18 Missouri         2  9
19 'Utah␣St'        1  12
20 UCLA             2  4
21 USC              3  6
22 'Boston␣College' 2  3
23 Iowa             1  7
24 Kentucky         2  2
25 Stanford         3  1
26 "St␣Joseph's"    1  9
27 Cincinnati       2  5
28 'Kent␣St'        1  13
29 'Georgia␣St'     1  11
30 Maryland         4  3
31 Georgetown       2  10
32 Hampton          1  15 ;
param: seedcost :=
1      25
2      21
3      18
4      15
5      12
6      10
7       8
8       6
9       5
10      4
11      3
12      2
13      1
14      1
15      1
16      1 ;
```

The answer to this dynamic programming problem is obtained from AMPL through manipulation of the input parameters, where the input parameter f is constructed so that it creates f(n,s) as the correct value of number of max wins from teams n..N using at most n pennies. While this works, it isnt the correct use of the parameter functionality in AMPL and is not intuitive. In comparison, the same problem can be easily computed in Python, without using any Pyomo functionality, as shown by the code below.

Python

```
seedcostdict = {1:25, 2:21, 3:18, 4:15, 5:12, 6:10, 7:8, 8:6, 9:5,
                10:4,11:3, 12:2, 13:1, 14:1, 15:1, 16:1}
tupledict= {1:('Michigan St',4,1), 2:('Fresno St',1,9),
            3:('Gonzaga',2,12), 4:('Indiana St',1,13),
            5:('Temple',3,11), 6:('Florida',1,3),
            7:('Penn St',2,7), 8:('North Carolina',1,2),
            9:('Illinois',3,1), 10:('Charlotte',1,9),
            11:('Syracuse',2,5), 12:('Kansas',2,4),
            13:('Notre Dame',1,6), 14:('Missisippi',2,3),
            15:('Butler',1,10), 16:('Arizona',5,2),
            17:('Duke',6,1), 18:('Missouri',2,9),
            19:('Utah St',1,12), 20:('UCLA',2,4),
            21:('USC',3,6), 22:('Boston College',2,3),
            23:('Iowa',1,7), 24:('Kentucky',2,2),
            25:('Stanford',3,1), 26:('St Josephs',1,9),
            27:('Cincinatti',2,5), 28:('Kent St',1,13),
            29:('Georgia St',1,11), 30:('Maryland',4,3),
            31: ('Georgetown', 2, 10), 32: ('Hampton', 1, 15)}

def name(team): return tupledict[team][0]
def wins(team): return tupledict[team][1]
def seed(team): return tupledict[team][2]
def seedcost(seed): return seedcostdict[seed]
def cost(team): return seedcost(seed(team))

teams=32
seeds=16
pennies=100

# dynamic programming algorithm
def pack2(team,pennies):
    Table= [[0 for x in range(pennies+1)] for x in range(team+1)]
    for i in range(team + 1):
        for w in range(pennies + 1):
            if i == 0 or w == 0:
                Table[i][w] = 0
            elif cost(i) <= w:
                Table[i][w]= max(wins(i)+Table[i-1][w-cost(i)],
                Table[i-1][w])
            else:
                Table[i][w] = Table[i - 1][w]
    return Table[team][pennies]

def rectab(team,pennies):
    Table= [[0 for x in range(pennies+1)] for x in range(team+1)]
    for i in range(team + 1):
        for w in range(pennies + 1):
            if i == 0 or w == 0:
                Table[i][w] = 0
            elif cost(i) <= w:
                Table[i][w]= max(wins(i)+Table[i-1][w-cost(i)],
                Table[i-1][w])
            else:
                Table[i][w] = Table[i - 1][w]
    return Table

result = pack2(32,100)
table=rectab(32,100)
print result
i=32
pen=100

while i>0 and pen-cost(i)>=0:
    if table[i][pen] == table[i-1][pen-cost(i)] + wins(i):
        print name(i)
        i=i-1
        pen=pen-cost(i)
    else:
        i=i-1
```

While the code does require the data to be inputted as python dictionaries, the logic in the construction of dynamic programming algorithm is very straightforward, and is similar to algorithms that students have seen in their introductory computer science courses.

Re-solving Models

Another large benefit in Pyomo is the ability to solve linear programs, and then add constraints or more data, and then resolve. In addition to that, it is easy to automate the solving and re-solving of models. An instance that this becomes useful is in the cutting stock problem. In the cutting stock problem, a key issue is that it is impossible to list all the cutting stock patterns that can be used and add all of those to the linear program. Instead, the best method of solving it is to solve the problem with a limited number of patterns, check if it is optimal, and if not, add more patterns and re-solve until an optimal solution is obtained. The process of checking whether the solution is optimal and finding more patterns to add requires solving a second linear program, a knapsack problem. Thus, the final solution is obtained by solving the cutting stock problem, solving a knapsack problem, re-solving the cutting stock problem, solving a knapsack problem, and so on. This is a key place where Python and Pyomo excel and work better than AMPL.

Pyomo

```
from pyomo.environ import *
from pyomo.opt import SolverFactory
from pyomo import *
from numpy import *

def cuttingstock(filename, addition):
    opt = SolverFactory('gurobi')
    model = AbstractModel()
    model.i = Set()
    model.j = Set()
    model.demand = Param(model.i)
    model.n = Param(model.i*model.j, mutable=True)
    model.decisionvar = Var(model.j, within=NonNegativeReals)

    def NetScore(model):
        return sum(model.decisionvar[i] for i in model.j)

    model.maxwins = Objective(rule=NetScore, sense=minimize)

    def constr(model, i):
        return sum(model.n[i,j]*model.decisionvar[j]
        for j in model.j) >= model.demand[i]

    model.deemand = Constraint(model.i, rule=constr)

    instance = model.create_instance(filename)
    results = opt.solve(instance)

    if addition!=[]:
        counter=2
        for i in addition:
            counter+=1
            instance.j.add(counter)
            instance.decisionvar.add(counter)
            instance.n[1,counter]=i[0]
            instance.n[2,counter]=i[1]
    instance.preprocess()
    instance.deemand.reconstruct()
    instance.del_component(instance.maxwins)
    instance.add_component("maxwins", Objective(rule=NetScore, sense=minimize))

    #the objective function can also be updated by overwriting it
    #instance.maxwins=Objective(rule=NetScore, sense=minimize)

    results = opt.solve(instance)

    instance.display()
    resultlist1=[]
    for v in instance.component_data_objects(Var):
        resultlist1.append(v.value)
    return resultlist1

def knapip(filename, addition):
    opt = SolverFactory('gurobi')
    model = AbstractModel()
    model.N = Param()
    model.W = Param()
    model.patterns = Param(RangeSet(1, model.N))
    model.value = Param(RangeSet(1, model.N), mutable=True)
    model.weight = Param(RangeSet(1, model.N))
    model.decisionvar = Var((RangeSet(1, model.N)), within=NonNegativeIntegers)

    def NetValue(model):
        return sum(model.decisionvar[i] * model.value[i]
                    for i in model.patterns)

    model.maxvalue = Objective(rule=NetValue, sense=maximize)

    def CostConstraint(model):
        return sum(model.decisionvar[i] * model.weight[i]
        for i in model.patterns) <= model.W

    model.maxweight = Constraint(rule=CostConstraint)

    instance = model.create_instance(filename)
    if addition!=[]:
        instance.value[1]=addition[0]
        instance.value[2]=addition[1]

    results = opt.solve(instance)
    instance.display()
    resultlist1=[]
    for v in instance.component_data_objects(Var):
        resultlist1.append(v.value)
    return (instance.maxvalue(), resultlist1)

def matrixmath(vector, othervec):
    lst = []
    counter=0
    for i in vector:
        if i>0:
            lst.append(othervec[counter])
        counter+=1
    invlst = linalg.inv(lst)
    varstore = invlst[0][1]
    invlst[0][1]=invlst[1][0]
    invlst[1][0]=varstore
    return dot([1.,1.], invlst)



if __name__ == "__main__":
    filename1= raw_input('Enter name of data file:')
    filename1= filename1 + '.dat'
    filename2= raw_input('Enter name of knapsack data file:')
    filename2= filename2 + '.dat'
    cuttinginput=[]
    cuttingoutput = cuttingstock(filename1, [])
    print cuttingoutput
    knapoutput=knapip(filename2, [])
    print knapoutput
    setofpatterns = [[16,0],[0,9]]
    while knapoutput[0]>1:
       cuttinginput.append(knapoutput[1])
       setofpatterns.append(knapoutput[1])
       cuttingoutput = cuttingstock(filename1, cuttinginput)
       print cuttingoutput
       print setofpatterns
       knapoutput = knapip(filename2,matrixmath(cuttingoutput,setofpatterns))
```

In this file, two python functions are defined. The first one takes in the name of an AMPL data file, as well as an additional list of cutting stock patterns, and solves the cutting stock problem on the parameters and data in the file along with additional patterns defined by the inputted list. This is an example where Pyomo has additional functionality over AMPL. While Pyomo can solve the cutting stock problem from the data in the AMPL data file, it can also take in data from other sources, incorporate that into the AMPL data, and solve the problem with both inputs. This uses a key feature of Pyomo that, while a bit clumsy in its implementation, makes Pyomo extremely useful. The first steps that happen within that function is that an abstract model is built with all the necessary parameters, variables, objective function and constraints, and then it is instantiated using the data in the AMPL data file. Once that instantiated model is created, Pyomo can edit the instance, adding extra constraints for the added patterns, and then solve the problem. The key to the model is that Python allows us to solve linear programs with added constraints or variables, and then return the result of that linear program, so that it can be manipulated. This model sequentially solves the cutting stock problem, and then feeds the solution to the cutting stock problem into the knapsack problem, which returns an additional constraint that must be added to the cutting stock problem. With the use of a while loop, this process can continue until the result reaches an optimal state, where in this case, the resulting objective function value of the knapsack problem is less than 1, meaning Pyomo has found the optimal solution to the cutting stock problem. Similar features can be used to add constraints, variables and mutate values of parameters for linear programming problems that require it. Finally, Pyomo, when returning the result of solving the linear program, can either display the values in an easy to read format, as shown below, or return the results as python data for further computation.

Pyomo Output

```
Variables:
  decisionvar : Size=2, Index=j
      Key : Lower : Value         : Upper : Fixed : Stale : Domain
        1 :     0 :        6.6875 :  None : False : False : NonNegativeReals
        2 :     0 : 8.66666666667 :  None : False : False : NonNegativeReals

Objectives:
  maxwins : Size=1, Index=None, Active=True
      Key  : Active : Value
      None :   True : 15.3541666667

Constraints:
  deemand : Size=2
      Key : Lower : Body  : Upper
        1 : 107.0 : 107.0 :  None
        2 :  78.0 :  78.0 :  None
```

Branch and Bound

Finally, Pyomo improves on AMPL by making it easier to demonstrate key features such as cutting planes and the branch and bound algorithm. The code below shows an implementation of the branch and bound algorithm for a simple Integer Program.

$$\begin{gathered}\max -x_1 + x_2 \\ s.t. - 10x_1 + 20x_2 \leq 22 \\ 5x_1 + 10x_2 \leq 49 \\ x_1 \leq 5 \\ x_1, x_2 \geq 0, integer\end{gathered}$$

Pyomo Dynamic Programming

```
from pyomo.environ import *
from pyomo.opt import SolverFactory
from pyomo import *
from numpy import *
from copy import *
from pyomo.opt import *

currentbest= (0,[0,0])

def branchandbound(addition):
    opt=SolverFactory('gurobi')
    N = [1, 2]
    M = 3
    c = {1: -1, 2: 4}
    a = {(1, 1): -10, (2, 1): 20, (1, 2): 5, (2, 2): 10, (1,3):1, (2,3):0}
    b = {1: 22, 2: 49, 3: 5}
    for i in addition[0][0]:
        M+=1
        a[(1,M)]=1
        a[(2,M)]=0
        b[M]=i
    for j in addition[0][1]:
        M+=1
        a[(1,M)]=0
        a[(2,M)]=1
        b[M]=j
    for i in addition[1][0]:
        M+=1
        a[(1,M)]=-1
        a[(2,M)]=0
        b[M]=-i
    for j in addition[1][1]:
        M+=1
        a[(1,M)]=0
        a[(2,M)]=-1
        b[M]=-j

    model = ConcreteModel()
    model.x = Var(N, within=NonNegativeReals)
    model.obj = Objective(expr=sum(c[i]*model.x[i] for i in N),sense=maximize)

    def con_rule(model, m):
        return sum(a[i, m] * model.x[i] for i in N) <= b[m]

    model.con = Constraint(RangeSet(M), rule=con_rule)
    results = opt.solve(model)

    if (results.solver.status == SolverStatus.ok) and (
        results.solver.termination_condition == TerminationCondition.optimal):
        print "feasible"
        model.display()
        resultlist1 = []
        for v in model.component_data_objects(Var):
            resultlist1.append(v.value)
        return (model.obj.expr(), resultlist1)
    else:
        print "infeasible"
        return (0,[0,0])


def branch(x,acc):
    global currentbest
    if int(x[1][0])==x[1][0] and int(x[1][1])==x[1][1]:
        if x[0]>=currentbest[0]:
            currentbest=deepcopy(x)
        return x
    else:
        if currentbest[0] >= x[0]//1:
            return (0,[0,0])
        acclow=deepcopy(acc)
        acchigh=deepcopy(acc)
        if int(x[1][0]) != x[1][0]:
            acclow[0][0].append(x[1][0]//1)
            acchigh[1][0].append((x[1][0]+1)//1)
        else:
            acclow[0][1].append(x[1][1]//1)
            acchigh[1][1].append((x[1][1]+1)//1)
        newxlow=branchandbound(acclow)
        newxhigh=branchandbound(acchigh)
        branchlow=branch(newxlow,acclow)
        branchhigh=branch(newxhigh,acchigh)
        if branchlow[0]>=branchhigh[0]:
            return branchlow
        else:
            return branchhigh


if __name__ == "__main__":
    x=branchandbound((([],[]),([],[])))
    print x
    y=branch(x,(([],[]),([],[])))
    print y
```

In this example, we are creating a concrete model – one that has all of its inputs already defined in the model, without the need to instantiate it with a data file. Because Pyomo is implemented in python, it has an advantage that AMPL does not. In this program, a function branchandbound is defined, where it takes in as input extra restrictions on each variable, either x_1 or x_2. The way the function is currently defined, with empty input, it will return the solution to the linear programming relaxation of the integer program. Given an input of say,

```
(([a,b],[c]),([],[d]))
```

Hello This implies that the constraints that x_1 $\leq$ a , x_1 $\leq$ b, x_2 $\leq$ c, and x_2 $\geq$ d must be added. This allows the user to first define the LP relaxation of the problem, and then add bounds when branching. Once this is defined, the rest can be done with a simple python function (named branch) that takes in the current values on the variables, and branches on the non-integer values and find the optimal integer solution, which is recorded in the global variable aptly named ”currentbest”.