ISyE/CS/Math 728: Integer Optimization
Software/Implementation

Prof. Luedtke

UW-Madison

Spring 2021

Outline

- Introduction to IP software in general
- Use of Gurobi with Python interface as an example

Integer Optimization Solvers

Commmercial solvers:

- Gurobi: Perhaps current leader
- CPLEX (Ilog, now owned by IBM): Long-time leader
- XpressMP (FICO): Close competitor
- Mosek: Better for continuous optimization

Non-commercial:

- Solving Constraint Integer Programs (SCIP): `scip.zib.de` – very good
- COIN Branch-and-cut (CBC): `www.coin-or.org`

Communicating instances to a solver

Before a solver can solve an integer optimization instance, you must tell it what it is!

How can we do this?

- Put the data into a text file in a "standard format"
  - You don't want to do this by hand!
- Formulate the model using an *Algebraic Modeling Language*
  - Examples: GAMS, AMPL, AIMMS, Pyomo, JuMP
  - This is what you should usually do
- Build the model using the interface of the solver you will use
  - Closest interaction with the solver
  - Easy to integrate solver with other code (C++, Python, ...)
  - CPLEX, Gurobi, Mosel, OPL

Using the solver API directly

API: Application Programming Interface

Build the model using interface functions supplied by the solver.

Advantages

- Access to all features of the solver
  - Parameters, Callbacks to change solver behavior
- Can use all power and generality of programming language (e.g. C,C++,Java,Python)
- Can be integrated into other applications

Disadvantages:

- Can take longer to write a model
- Married to the solver
- You can "break" a solver

Algebraic Modeling Langues (e.g., GAMS, AIMMS, JuMP) are adding more and more of the API functionality

In this course

You can use any solver you have access to. Possibilities:

- I'll introduce Gurobi, using a Python interface
  - You can also download a version for academic use: `http://www.gurobi.com`
- CPLEX: You may also be able to get an academic version
  - Can use callable library (C), Concert Technology (C++), or Python interface
- JuUMP, GAMS or AMPL – I won't be able to help much
- Pyomo, Coin CBC, or SCIP (if you are ambitious)
  - But you're really on your own there

Gurobi and python

- Gurobi: A MIP Solver.
- Python: Powerful general-purpose scripting language.

How does it work?

- Write a python script that builds and runs a model, using a Gurobi module
  - Can write script in any editor or an Integrated Development Environment (IDE)
- Run the script from a terminal: using gurobi.sh (in mac/linux) or gurobi.bat (in windows), or using your native python installation

Getting started

1. Register for a Gurobi account (use your `wisc.edu` e-mail)
2. Gurobi's quick start guide: `https://www.gurobi.com/documentation/quickstart.html`
3. Download and install Gurobi (painless)
4. Request then get a license – Gurobi quickstart documentation explains this well

Optional (maybe): Using your own python installation

Gurobi provides good online instructions for installing within Anaconda, and using e.g., Jupyter notebook or Spyder IDE for editing

- I will use the Spyder IDE

First example: mip1.py

Let's model a very simple integer program:

$$
\begin{aligned}
\max\ & x + y + 2z \\
\text{s.t. } & x + 2y + z \leq 4 \\
& x + y \geq 1 \\
& x, y, z \in \{0, 1\}
\end{aligned}
$$

Source: Gurobi quick start guide

Example mip1.py

```
import gurobipy as gp
from gurobipy import GRB

try:

    # Create a new model
    m = gp.Model("mip1")

    # Create variables
    x = m.addVar(vtype=GRB.BINARY, name="x")
    y = m.addVar(vtype=GRB.BINARY, name="y")
    z = m.addVar(vtype=GRB.BINARY, name="z")
```

Example mip1.py (cont'd)

```
    # Set objective
    m.setObjective(x + y + 2 * z, GRB.MAXIMIZE)

    # Add constraint: x + 2 y + 3 z <= 4
    m.addConstr(x + 2 * y + 3 * z <= 4, "c0")

    # Add constraint: x + y >= 1
    m.addConstr(x + y >= 1, "c1")

    # Optimize model
    m.optimize()

for v in m.getVars():
        print('%s %g' % (v.varName, v.x))

    print('Obj: %g' % m.objVal)
```

Gurobi attributes

Much of the interaction with Gurobi is done through attributes of Gurobi objects

- Gurobi Model attributes: numconstrs, numvars, modelname, runtime, itercount, nodecount, objval, ...
- Gurobi Var (variable) attributes: lb, ub, obj, vtype, varname, x (current solution value), rc (reduced cost), ...
- Gurobi Constr (constraint) attributes: sense, rhs, pi (dual variable value), slack, constrname, ...

Some attributes can only be queried, others can be queried or set

A little about python

Key python data structure: List

- A collection of arbitrary objects
- Can add to list, remove elements, change elements, etc.

```
l = [1, 'a', 3.5]
print l[1]  # prints 'a'
l += 'b'
# Now l = [1, 'a', 3.5, 'b']
```

A similar data structure: Tuple

- Main difference: Tuples cannot be changed once created
- Importance: Tuples can be used as *keys* to a dictionary

```
t = (1, 'a', 3.5)
print t[1]  # prints 'a'
t += 'b'  # Produces an error
```

A little about python (2)

Another data structure: Dictionary

- A collection of (key, object) pairs
- Like a list, except access via key rather than by index
- Useful for storing model elements, like decision variables

```
d = {}  # creates an empty dictionary
d[(5,2,3)] = 'object 1'  # Tuple (5,2,3) is the key

# parentheses on the tuple are optional
d[2,3,4] = 'object 2'  # Tuples (2,3,4) is the key
```

Python dictionary initialization

Python dictionaries can also be initialized like this:

```
values = { ‘zero’: 0, ‘one’: 1, ‘two’: 2 }
print values[‘zero’]  # prints 0
```

Gurobi provides a function, ‘multidict’, for simultaneously initializing multiple dictionaries having the same key set

```
names, lower, upper = multidict({ ‘x’: [0, 1],
                                  ‘y’: [1, 2],
                                  ‘z’: [0, 3] })
```

- Returns a list of the keys, followed by the dictionaries

Python list comprehension and the tuplelist class

Example of list comprehension for creating a list of tuples:

```
print [(x,y) for x in range(3) for y in range(3) if x != y]
# prints:  [(0, 1), (0, 2), (1, 0), (1, 2) (2, 0), (2, 1)]
# the range(i) command produces list [0,...,i-1]
```

Gurobi provides the ‘tuplelist’ class for efficiently selecting sublists of tuples

```
l = tuplelist([(1, 2), (1, 3), (2, 3), (2, 4)])
print l.select(1,‘*’)
# prints: [(1,2), (1,3)]
print l.select(‘*’,3)
# prints: [(1,3), (2,3)]
```

Gurobi's tupledict class

Similar to 'tuplelist', Gurobi provides a subclass of the general python dictionary called 'tupledict'

- Works like python dictionary, except that the keys are a tuplelist
- Enables efficient construction of linear expressions when variables are stored in a 'tupledict' object

```
l = list([(1, 2), (1, 3), (2, 3), (2, 4)]) # list of tuples
d = model.addVars(l, name="d") # creates a variable for every tuple in
                   # stores in tupledict object d
model.update()
sum(d.select(1, '*')) # creates an expression that sums over
                      # variables with first index = 1
d.sum(1,'*') # shortcut equivalent to the above statement
```

Example model: Multicommodity flow (netflow.py)

Mathematical formulation:

- Graph with nodes V and directed arcs A
- Set of commodities H
- Arc capacities b_{ij} for $(i,j) \in A$
- Unit shipment costs: c_{ijh} for $h \in H$, $(i,j) \in A$
- Supply/demand of each commodity at each node: s_{ih} for $i \in V$, $h \in H$

Decision variables:

- x_{ijh}: flow of commodity $h \in H$ on arc $(i,j) \in A$

Objective:

$$\min \sum_{(i,j) \in A} \sum_{h \in H} c_{ijh} x_{ijh}$$

Example model: Multicommodity flow (netflow.py)

Arc capacity constraints:

$$\sum_{h \in H} x_{ijh} \leq b_{ij}, \quad \forall (i,j) \in A$$

Flow balance constraints:

$$\sum_{(i,j) \in A} x_{ijh} + s_{jh} = \sum_{(j,i) \in A} x_{jih}, \quad \forall j \in V, h \in H$$

Gurobi python model (netflow.py)

```
import gurobipy as gp
from gurobipy import GRB

# Base data
commodities = ['Pencils', 'Pens']
nodes = ['Detroit', 'Denver', 'Boston', 'New York', 'Seattle']

arcs, capacity = gp.multidict({
    ('Detroit', 'Boston'):   100,
    ('Detroit', 'New York'):  80,
    ('Detroit', 'Seattle'):  120,
    ('Denver',  'Boston'):   120,
    ('Denver',  'New York'): 120,
    ('Denver',  'Seattle'):  120})
```

Gurobi python model (netflow.py)

Initializing the cost data dictionary

```
# Cost for triplets commodity-source-destination
cost = {
    ('Pencils', 'Detroit', 'Boston'):   10,
    ('Pencils', 'Detroit', 'New York'): 20,
    ('Pencils', 'Detroit', 'Seattle'):  60,
    ('Pencils', 'Denver',  'Boston'):   40,
    ('Pencils', 'Denver',  'New York'): 40,
    ('Pencils', 'Denver',  'Seattle'):  30,
    ('Pens',    'Detroit', 'Boston'):   20,
    ('Pens',    'Detroit', 'New York'): 20,
    ('Pens',    'Detroit', 'Seattle'):  80,
    ('Pens',    'Denver',  'Boston'):   60,
    ('Pens',    'Denver',  'New York'): 70,
    ('Pens',    'Denver',  'Seattle'):  30}
```

Gurobi python model (netflow.py)

Initializing the flow data dictionary

```
# Demand for pairs of commodity-city
inflow = {
    ('Pencils', 'Detroit'):   50,
    ('Pencils', 'Denver'):    60,
    ('Pencils', 'Boston'):   -50,
    ('Pencils', 'New York'): -50,
    ('Pencils', 'Seattle'):  -10,
    ('Pens',    'Detroit'):   60,
    ('Pens',    'Denver'):    40,
    ('Pens',    'Boston'):   -40,
    ('Pens',    'New York'): -30,
    ('Pens',    'Seattle'):  -30}
```

Gurobi python model (netflow.py)

Creates the Gurobi model object and the flow variables

```
# Create optimization model
m = gp.Model('netflow')

# Create variables
flow = m.addVars(commodities, arcs, obj=cost, name="flow")

# Arc-capacity constraints
m.addConstrs(
    (flow.sum('*', i, j) <= capacity[i, j] for i, j in arcs), "cap")


# Equivalent version using Python looping
# for i, j in arcs:
#   m.addConstr(sum(flow[h, i, j] for h in commodities) <= capacity[i,
#                   "cap[%s, %s]" % (i, j))
```

Gurobi python model (netflow.py)

Creates the flow balance constraints

```
# Flow-conservation constraints
m.addConstrs(
    (flow.sum(h, '*', j) + inflow[h, j] == flow.sum(h, j, '*')
        for h in commodities for j in nodes), "node")

# Alternate version:
# m.addConstrs(
#   (gp.quicksum(flow[h, i, j] for i, j in arcs.select('*', j)) + inflow[h, j]
#     gp.quicksum(flow[h, j, k] for j, k in arcs.select(j, '*'))
#     for h in commodities for j in nodes), "node")
```

- the ‘select’ function is used for the tuplelist arcs to efficiently choose the right arcs to sum over
- ‘quicksum’ is a Gurobi function that creates the expression representing the sum of the decision variables

Gurobi python model (netflow.py)

Solve the model, check the status, and display the solution

```
# Compute optimal solution
m.optimize()

# Print solution
if m.status == GRB.OPTIMAL:
    solution = m.getAttr('x', flow)
    for h in commodities:
        print('\nOptimal flows for %s:' % h)
        for i, j in arcs:
            if solution[h, i, j] > 0:
                print('%s -> %s: %g' % (i, j, solution[h, i, j]))
```

- Solution value is stored in the attribute 'x' of the flow variable object
- NB: Obtaining all variable values with one call is *much* more efficient than looping and obtaining one at a time

Solver parameters

MIP solvers have many parameters that you can change to possible change the performance

- E.g., many of the choices in branch-and-bound we saw

In Gurobi's python interface, change a parameter associated with a model 'm' with the commands:

- m.setParam("paramname", paramvalue)
- m.Params.paramname = paramvalue

where 'paramname' and 'paramvalue' are placeholders

Examples of paramaters

Nearly all solvers have similar sets of parameters

- MIPFocus: Feasible solutions (1), optimality (2), or bound (3)
- ImproveStartTime: After this amount of time, focus entirely on finding better solutions
- TimeLimit: Stop after this amount of time has elapsed
- MIPGap: Stap when gap between lower and upper bound reaches this value
- NodeLimit: Stop after processing this many nodes
- Method: Chooses method for solving the initial LP relaxation
- Cuts: Aggressive (2), Conservative (1), Automatic (-1), or None (0)
- FlowCoverCuts, etc.: Aggressive (2), Conservative (1),...
- VarBranch: Change the branching variable selection strategy
- Presolve: Aggressive (2), Conservative (1),...

More Gurobi examples

Check out additional examples on your own

- dietmodel.py and diet2.py: Python function, separate data from model
- fixanddive.py: Implements a simple MIP-based heuristic

Two types of cuts

User cuts: These never cut off an integer solution that is feasible to the formulation loaded to the solver

- These are valid inequalities in the traditional sense
- Only purpose of these cuts is to improve relaxation bound

Lazy cuts: These inequalities help to *define the feasible region*

- They might cut off an integer solution feasible to the formulation originally loaded to the solver
- Example: Subtour elimination constraints in TSP
- Solvers will let you attempt to generate lazy cuts *at every integer solution*

You must tell Gurobi which type of cut you are adding.

Using lazy cuts

- Using lazy cuts to solve a problem with exponentially constraints is sometimes referred to *delayed constraint generation*
- If you plan to do this, you must turn off some presolve reductions
  - These reductions are based on feasibility considerations of the problem, so are not valid if you will change the feasible region
- In Gurobi: m.Params.lazyConstraints = 1

Gurobi callbacks

First, declare and implement callback function like this:

```
def mycallback(model, where):
   # Implement the callback routine here
```

Instruct Gurobi to use the callback when you optimize the model:

```
m.optimize(mycallback)
```

If you need to access data or objects from within the callback, you may add them as private members of the model object, e.g.,

```
m._vars = vars
```

- vars is a list created earlier in the code
- m._vars then stores a pointer to that list for use in the callback
- Names of user data must begin with an underscore

Works but "bad practice": declare global variables above callback function

Within the callback function

Gurobi provides special functions that can be called within the callback function

- model.cbGetSolution, model.cbGetNodeRel, model.cbLazy, model.cbCut, model.cbGet, model.cbSetSolution

The *where* parameter of the callback routine tells you what step of the solution process Gurobi is in, and limits which functions you can call

```
def mycallback(model, where):
   if where == GRB.callback.SIMPLEX:
      # maybe query something
   if where == GRB.callback.MIP:
      # maybe query something
   if where == GRB.callback.MIPNODE:
      # maybe add user cuts, or set a heuristic solution
   if where == GRB.callback.MIPSOL:
      # maybe add lazy cuts
```

Getting information in the callback function

Most information on current status can be queried with the function:

- mode.cbGet(what)
- 'what' is the code for the information desired
- Allowed values for 'what' depend on the value of 'where'

```
def mycallback(model, where):
   if where == GRB.callback.SIMPLEX:
      print model.cbGet(GRB.callback.SPX_ITRCNT)
if where == GRB.callback.MIPSOL:
print model.cbGet(GRB.callback.MIPSOL_OBJ)
```

Gurobi Example: tsp.py

This example finds and adds subtour elimination constraints *only* at integer solutions

```
def subtourelim(model, where):
    if where == GRB.Callback.MIPSOL:
        # make a list of edges selected in the solution
        vals = model.cbGetSolution(model._vars)
        selected = gp.tuplelist((i, j) for i, j in model._vars.keys()
                                if vals[i, j] > 0.5)
        # find the shortest cycle in the selected edge list
        tour = subtour(selected)
        if len(tour) < n:
            # add subtour elimination constr. for every pair of cities
            model.cbLazy(gp.quicksum(model._vars[i, j]
                                     for i, j in combinations(tour, 2))
                         <= len(tour)-1)
```

Possible variation: Also search for SEC's at *fractional* solutions