gurobi 可视化建模环境

— python 编程

刃之砺运筹学优化部
weibo.com/edgestone

刃之砺信息科技（上海）有限公司
info@edgestone-it.com

直译式程序设计语言—python

- python是一种功能强大的通用型语言，成熟且稳定。语法简捷和清晰，能够轻松完成很多常见的任务；
- python的设计哲学是“优雅”、“明确”、“简单”；
- python开发者的哲学是“用一种方法，最好是只有一种方法来做一件事”；
- python直接编写的程序段有时运行效率甚至高于用C编写的程序；
- 在Google内部的很多项目使用C++编写性能要求极高的部分，然后用python调用相应的模组；
- YouTube、Google、Yahoo!、NASA都在内部大量地使用python。

http://zh.wikipedia.org/wiki/Python（维基百科）
http://en.wikipedia.org/wiki/Python_%28programming_language%29
http://baike.baidu.com/view/21087.htm （百度百科）
http://groups.google.com/group/python-cn?hl=zh-CN （python 中文社区）

功能强大的 python

- **简单**：python是一种代表简单主义思想的语言。阅读一个良好的python程序就感觉像是在读英语一样，它使你能够专注于解决问题而不是去搞明白语言本身；
- **易学**：python极其容易上手，因为python有极其简单的语法；
- **免费、开源**：python 是 FLOSS（自由/开放源码软件）之一，是可以被移植在许多平台上的高层语言；
- **可扩展性和可嵌入性**：编写的系统管理脚本在可读性、性能、源代码重用度、扩展性几方面都优于普通的shell脚本；
- **丰富的模块**：可以进行科学计算编程，可以快速开发桌面应用编程等，能够支撑大规模的软件开发；
- **规范的代码**：python采用强制缩进的方式使得代码具有较好可读性。

http://zh.wikipedia.org/wiki/Python（维基百科）
http://en.wikipedia.org/wiki/Python_%28programming_language%29
http://baike.baidu.com/view/21087.htm （百度百科）
http://groups.google.com/group/python-cn?hl=zh-CN （中文社区）

python 中的基本操作和数据结构

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
>>> word = 'help' + 'A'
>>> word
'helpA'
>>> a=['spam','eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
>>> print [(i, i*i) for i in range(1,10) if i%2==0 ]
[(2, 4), (4, 16), (6, 36), (8, 64)]
>>>
```

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.25), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

更多关于 python 的介绍，请参阅下面链接：

http://docs.python.org/release/2.7.2/tutorial/datastructures.html

http://docs.python.org/release/2.7.2/tutorial/introduction.html

file:///D:/gurobi500/win32/docs/quickstart/node13.html

python 中的数据结构

列表（List）、元组（Tuples）、与字典（Dictionary）都是 python 中自带的数据结构。为了在gurobi 中更高效地建立抽象模型，gurobi 的 python 建模环境中还提供了一个客户化的数据结构 TupleList 和一些客户化函数，使得 gurobi python 足够强大和灵活，但又非常简洁和自然。

list — 列表， tuples — 元组

- [1, 2, 3]， ['Pens', 'Denver', 'New York'] 表示是列表；
- (1, 2, 3)， ('Pens', 'Denver', 'New York')表示是元组。
- 共同点：有序集合，通过偏移读取。
- 不同点：元组是无法修改的，这意味着一旦创建了元组后就不能再对其进行修改。在列表中，我们则可以随意添加、删除、或修改其中的成员。
- 我们可以利用元组的这种特性将其作为字典的索引（下标）来使用。

```
>>> list_l=[1, 0.618, 'EdgeStone']
>>> tuples_t=(1, 0.618, 'EdgeStone')
>>> print list_l[0], list_l[2]
1 EdgeStone
>>> print tuples_t, tuples_t[1]
(1, 0.618, 'EdgeStone') 0.618
>>>
```

dictionary—字典

dict ={ 'EdgeStone': 'Resellor', 'Gurobi': 'Optimal Engine', 'Python': 'Programming'}

特点

1.以键值对的方式存在并操作；2.通过键来存取，而非偏移量；3.键值对是无序的；4.键和值可以是任意对象；5.长度可变，任意嵌套；6.在字典里，不能再有序列操作，虽然字典在某些方面与列表类似，但不要把列表套在字典上；7.任何无法修改的 python 对象都可以当作键值使用：整数、浮点数、字符串、甚至于元组。

```
>>> dict={'EdgeStone': 'Resellor', 'Gurobi':
      'Optimal Engine', 'Python': 'Programming'}
>>> print dict
{'Python': 'Programming', 'Gurobi': 'Optimal Engine', 'EdgeStone': 'Resellor'}
>>> print dict['Gurobi']
Optimal Engine
>>> dict2={}
>>> for i in range(1,5):
        for j in range(i+1,5):
                if i*i<=j:
                        dict2[i,j]='EdgeStone'
                        print 'dict2[%d,%d]=%s'%(i,j,dict2[i,j]),
        print


dict2[1,2]=EdgeStone dict2[1,3]=EdgeStone dict2[1,4]=EdgeStone
dict2[2,4]=EdgeStone
```

multidict — 多个字典

name, property1 = multidict{(‘EdgeStone‘, ’Resellor’): 1, (‘Gurobi‘, ’Optimal Engine‘): 2, ('Python‘,’ 'Programming'): 3}

特点：可以在一条语句中初始化一个或多个字典对象。该函数的输入参数为一个字典对象，其中每一个键值都对应着一个长度为 n 的列表。该函数会将其中每一个列表都拆分成 n 个单项，从而创建出 n 个独立的字典。函数的返回值是一个列表，其中第一个结果是共享的键值列表，后面紧跟着的是新创建的 n 个字典对象。

```
>>> name, property1 = multidict({
        ('EdgeStone', 'Resellor'): 1,
        ('Gurobi', 'Optimal Engine'): 2,
        ('Python','Programming'): 3})
>>> print name
[('Gurobi', 'Optimal Engine'), ('EdgeStone', 'Resellor'), ('Python', 'Programming')]
>>> print property1
{('Gurobi', 'Optimal Engine'): 2, ('EdgeStone', 'Resellor'): 1, ('Python', 'Programming'): 3}
>>> print property1['EdgeStone','Resellor']
1
>>>
```

tuplelist—自定义类

- tuplelist 是 gurobi 在python 建模环境中自定义的一个类使用它能够有效地在元组列表中检索符合要求的子列表.具体来说，借助 tuplelist 对象中的 select 方法，可以返回所有满足特定条件（使用特定字段来匹配一个或多个特定值）的元组子集；
- select 方法可以通过一些内部数据结构来提高数据选择的效率；
- tuplelist 是列表类的一个子类，因此您可以使用标准的列表方法来访问或修改一个tuplelist 对象。

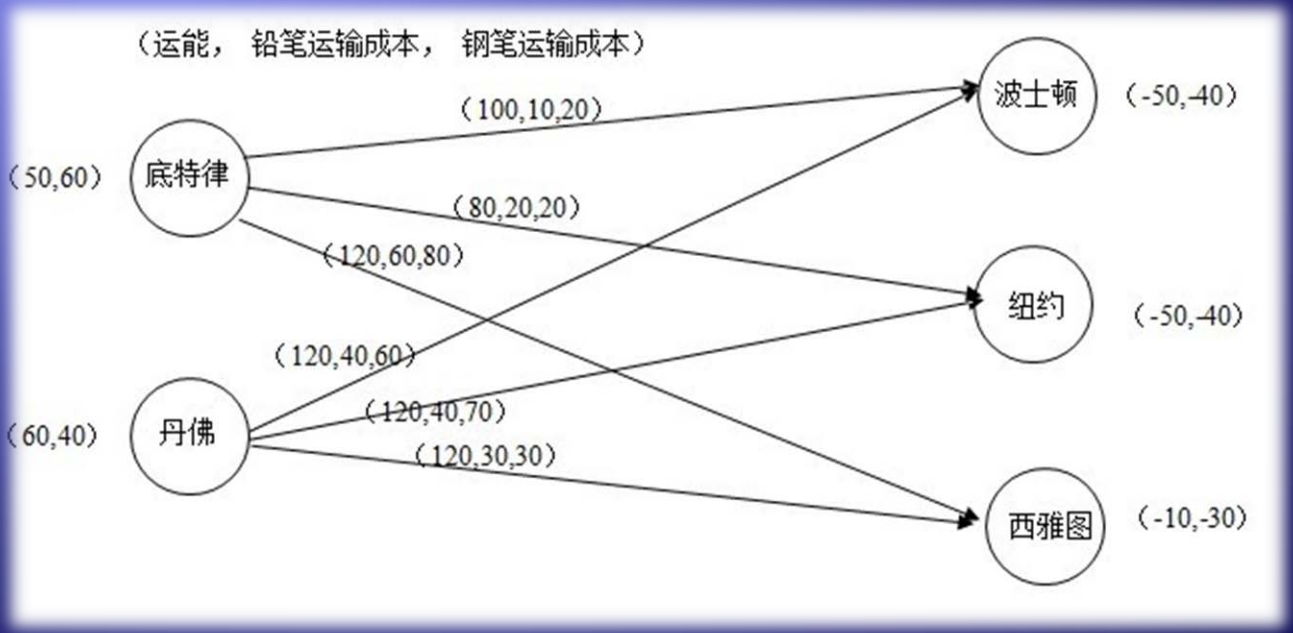
```
>>> name=tuplelist(name)
>>> type(name)
<class 'gurobipy.tuplelist'>
>>> print name.select('EdgeSeone','*')
[]
>>> print name.select('EdgeStone','*')
[('EdgeStone', 'Resellor')]
```

挑战

- 任何一个优化模型中最为重要的部分总是它所包含的那些决策变量。在实现模型时经常会遇到的一个主要的挑战就是：如何才能方便地存储并访问这些变量。尽管有些模型变量可以通过编程语言中简单的数据结构十分自然地进行表示（例如，x[i]与连续整数值 i），然而在大多数其他的模型中，变量的表示可能会复杂许多。例如，我们考虑一个这样的模型，它用于优化供应网络中多种不同商品的流量。您也许会定义一个变量 x['Pens', 'Denver', "New York"] 来表示某一种制造产品（本例中为钢笔）从丹佛运输到纽约的数量。与此同时，您也许并不需要变量 x['Pencils', "Denver", "Seattle"]，因为并不是所有的商品、输出城市、与目标城市的组合都是有效的网络路径。在普通的编程语言中，决策变量的稀疏集合表示通常是十分麻烦的。
- 除此之外，您通常还会在模型中添加一些约束。约束中须要包含决策变量的某些子集，这无疑进一步增加了建模的难度。以我们之前提到的网络流量模型为例，您也许想要设定一个上界值，用于限制流入某个目标城市的商品总量。您当然可以遍历每一个城市，并收集所有表示从这个城市进入目标城市的商品流量。然而，如果并非所有的输出城市 - 目标城市组合都是有效的，那么这种做法显然将会十分的低效。在一个庞大的网络问题中，这种低效的方法有可能会导致严重的效率问题。通常来说，使用前面介绍的较为复杂的数据结构可以有效地解决这一问题。

案例—网络流问题

- 我们以一个相对复杂的案例作为介绍。如果用其它编程语言来实现这个模型，大概需要几百行的程序代码，而用gurobi python仅仅只需要几十行。
- 有两种商品（铅笔与钢笔），它们在两个城市（底特律与丹佛）中进行生产，然后须要运送到位于另外三个城市（波士顿、纽约、与西雅图）的仓库中以满足给定的需求。运输网络中的每一条路径都有与之相对应的运输成本以及总运能。



网络流数学模型

$$\min \sum_{h\in commodities} \sum_{i,j\in arcs} flow[h,i,j]\times cost[h,i,j]$$

$$s.t. \sum_{h\in commodities} flow[h,i,j] \le capacity[i,j],\ \forall\ i,j\ \in arcs$$

$$\sum_{(i,j)\in arcs('*',j)} flow[h,i,j]+inflow[h,j]= \sum_{(j,k)\in arcs(j,'*')} flow[h,j,k],\ \forall \begin{cases} h \in commodities \\ j \in nodes \end{cases}$$

$$0 \le flow[h,i,j] \le capacity[i,j],\ \text{is integer variable}\ \forall\ i,j\ \in arcs$$

flow[h,i,j]	商品 h 从城市 i 到城市j 的运输数量（变量）
cost[h,i,j]	商品 h 从节点 i 到城市j 的运输成本（常量）
capacity[i,j]	从城市i 到城市j 的运能（常量）
arcs	所有可能的运输线路（常量）
nodes	所有可能的运输线路上的城市（常量）
commodities	所有等待运输的产品（常量）

初始化数据

```
commodities = ['Pencils', 'Pens']
nodes = ['Detroit', 'Denver', 'Boston', 'New York', 'Seattle']

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

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 }

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 }
```

创建模型

```
# 创建优化模型
m = Model('netflow')

# 创建决策变量
flow = {}
for h in commodities:
    for i,j in arcs:
        flow[h,i,j] = m.addVar(ub=capacity[i,j], obj=cost[h,i,j],
                               name='flow_%s_%s_%s' % (h, i, j))
m.update()

# 运能约束
for i,j in arcs:
    m.addConstr(quicksum(flow[h,i,j] for h in commodities) <= capacity[i,j],
                'cap_%s_%s' % (i, j))

# 能量守恒约束
for h in commodities:
    for j in nodes:
        m.addConstr(
            quicksum(flow[h,i,j] for i,j in arcs.select('*',j)) +
                inflow[h,j] ==
            quicksum(flow[h,j,k] for j,k in arcs.select(j,'*')),
                    'node_%s_%s' % (h, j))
```

给netflow创建一个模型，并把它传给m

给模型m 添加变量，并把它存在flow中

给模型m 添加约束

给模型m 添加约束

添加变量

- 在 gurobi 的 python 建模环境中，大多数行为都是通过调用 gurobi 对象的方法而执行的。
- **addVar (lb=0.0, ub=GRB.INFINITY, obj=0.0, vtype=GRB.CONTINUOUS, name="", Column=None)**

  lb（可选）：新变量的下界值；

  ub（可选）：新变量的上界值；

  obj（可选）：新变量的目标系数；

  vtype（可选）：新变量的变量类型（GRB.CONTINUOUS、GRB.BINARY、GRB.INTEGER、GRB.SEMICONT、或 GRB.SEMIINT）；

  name（可选）：新变量的名称；

  column（可选）：Column 对象，指定包含新变量的一组约束、以及相关联的系数。
- 返回值：新的变量对象。
- 应用示例：

```
x = model.addVar()
y = model.addVar(vtype=GRB.INTEGER, obj=1.0, name="y")
z = model.addVar(0.0, 1.0, 1.0, GRB.BINARY, "z")
```

添加约束

- addConstr (lhs, sense, rhs, name="")
- 输入参数：

  lhs：新线性约束的左侧端。可以是一个常量、一个Var、或一个LinExpr。

  sense：新线性约束的含义（GRB.LESS_EQUAL、GRB.EQUAL、或GRB.GREATER_EQUAL）。

  rhs：新线性约束的右侧端。可以是一个常量、一个Var、或一个LinExpr。

  name：新约束的名称。
- 返回值：新的约束对象。
- 应用示例：

```
lhs = LinExpr([1.0, 1.0], [x, y])   # x + y <= 2.0
model.addConstr(lhs, GRB.LESS_EQUAL, 2.0, "c0")
model.addConstr(x + 2 * y + 3 * z >= 4, "c0")
```

gurobi 求解

```
m.optimize()

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

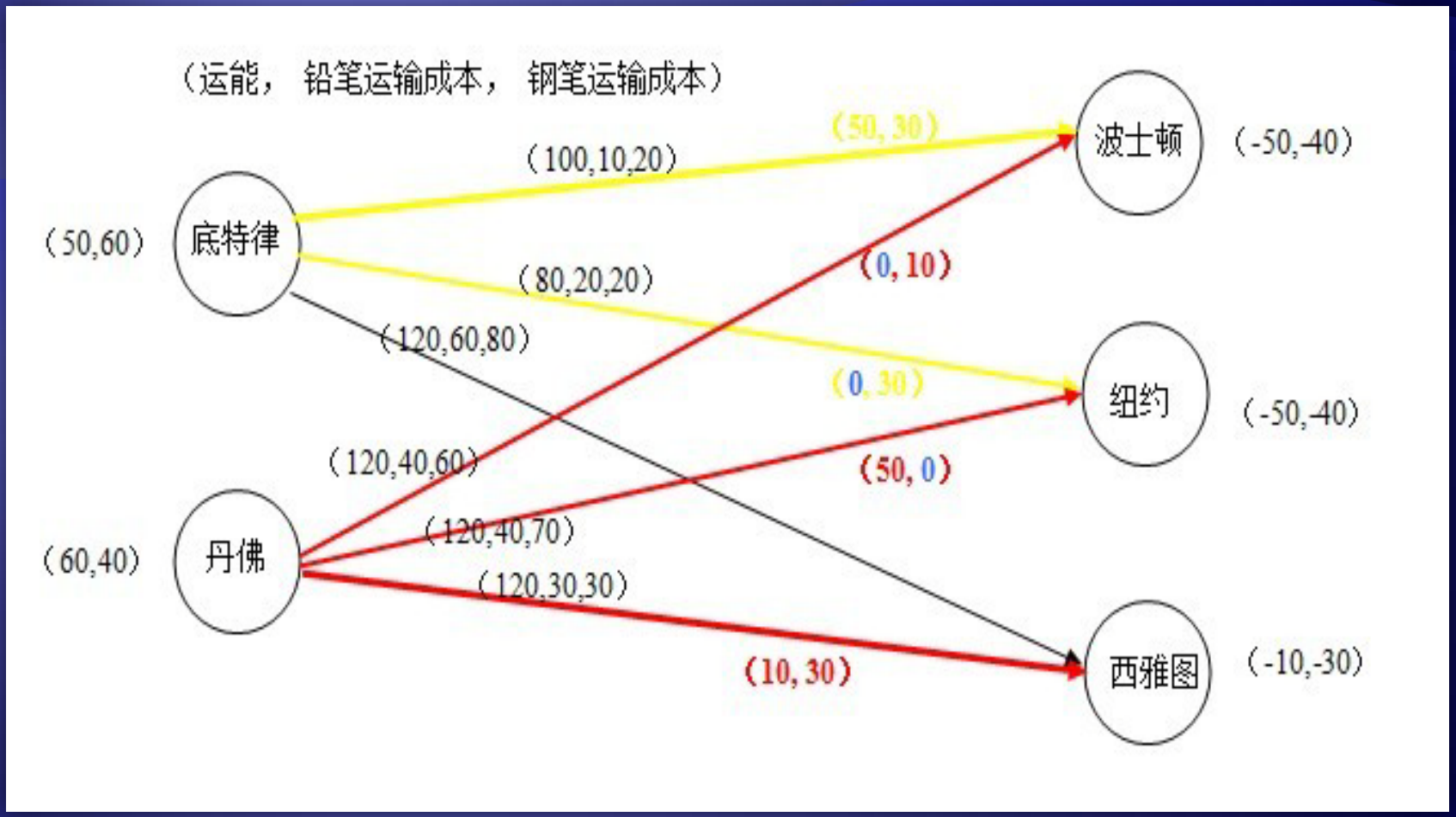
如果问题最优解已经被找到，则依照用户指令输出最优解和最优值

```
Optimal flows for Pencils :
Denver -> Seattle : 10.0
Denver -> New York : 50.0
Detroit -> Boston : 50.0

Optimal flows for Pens :
Denver -> Seattle : 30.0
Detroit -> New York : 30.0
Detroit -> Boston : 30.0
Denver -> Boston : 10.0
```

```
Optimize a model with 16 rows, 12 columns and 36 nonzeros
Presolve removed 16 rows and 12 columns
Presolve time: 0.00s
Presolve: All rows and columns removed
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    5.5000000e+03   0.000000e+00   0.000000e+00      0s

Solved in 0 iterations and 0.00 seconds
Optimal objective  5.500000000e+03
```

惊人的求解速度

最优运输策略


（运能， 铅笔运输成本， 钢笔运输成本）
（50,60）
底特律
（60,40）
丹佛
波士顿
（-50,-40）
纽约
（-50,-40）
西雅图
（-10,-30）
（100,10,20）
（80,20,20）
（120,60,80）
（120,40,60）
（120,40,70）
（120,30,30）
（50, 30）
（0, 10）
（0, 30）
（50, 0）
（10, 30）