UNIVERISTY OF MISSOURI, COLUMBIA

Computational Optimization Exam Answers

Nguyen Ph. Nguyen

December 1st, 2014

Abstract

This paper presents method to apply computer in Computational Optimization. The concept here is combination of functional software for sub processes in a whole workflow. The tasks done in sub processes are from preparing data, models and configuration to solving problem, and final evaluation. This can benefits research project in productivity and support experiment design.

1 Introduction of research workflow in Computational Optimization

Computational Optimization applies computers to solve optimization problems. These problems can be supply chain optimization, transportation planning problem, or energy efficiency problem that are usually large and complicated. Expected results are not only solutions but also performance. Researchers need to formulate problems, collect/process data, build models, run and evaluate results. This work usually requires serial and/or parallel combination of many tasks in many software programs.

Each program has specific function (data processing data analyzing, solving ...), is written in some language (Java, Python, C++ ...), and run in some platform (Windows, Linux, OsX). One alternative for this is using API to directly connect between 2 programs (such as r2py ...). The advantages of this method is faster speed. However, when the number of program increases,

communication in this network become too complicated and heavy. We need a flexible solution that separates data part and program part in our job. Each program will work with database to exchange information with other programs via database. A workflow will control and monitor all these activities: generating, collecting, storing, analyzing and visualizing data and results. That's the automating research workflow.

It also uses configuration (.ini) file to allow users to flexibly change working scenarios or configuration. With this kind of workflow, automation, handling large amount of data with high accuracy, and easy reproduction, flexibly changing scenarios are advantages to increase research productivity.

One important thing is automating workflow supports to do computational experiment in research process. Workflow helps to generate, collect, storing, analyze and visualize data and results. So now researchers can fully focus on their reasoning work, easily change, or combine or reproduce study scenarios, or simulate to find the best solution.

2 Typical research workflow in Computational Optimization

So the tasks are ordered in a workflow to be automatically implemented by an orchestration. There are 2 types of task in workflow: independent task and dependant task. Independent tasks can run simultaneously, while dependant tasks must run in order. A typical combination of workflow is diamond workflow, shown in Figure 1. Task 2 and Task 3 are independent, and both of them depend on Task 1.

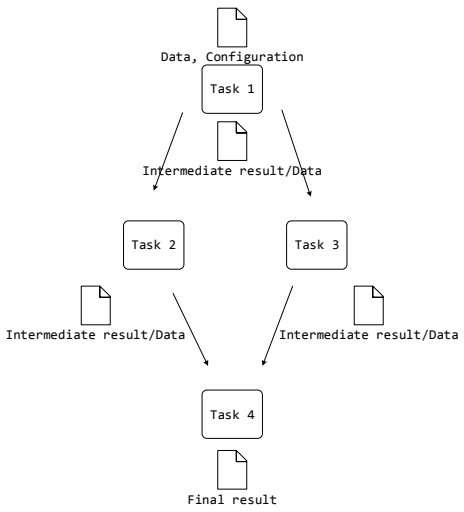

Figure 1: Diamond workflow

A specialized case of diamond workflow in Computational Optimization is shown in Figure 2. In this workflow, Task 1, 2, and 3 are for preparation. Data are collected, cleaned, and stored in csv file for database (SQLite) and passed to workflow.

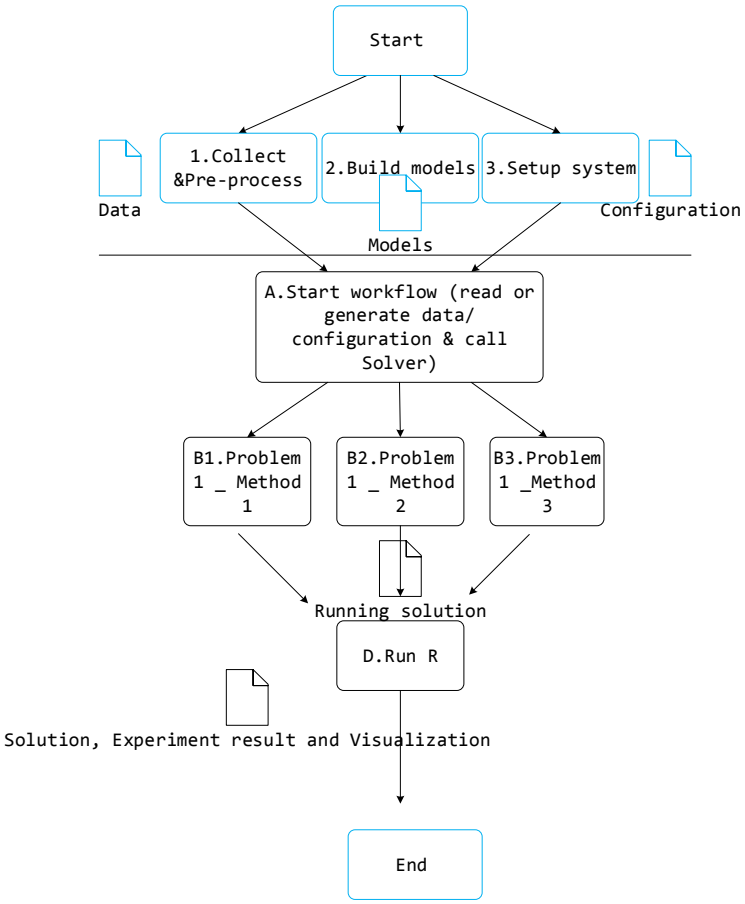

Figure 2: Typical workflow in Computational Optimization

Real automating workflow begins from Task A, reads configuration file (.ini) to initialize path (data, program), environment variable, and reads data. After orchestration is ready, Task A will pass data and models to solvers. The solvers here can be Gurobi, Cplex, Microsoft Foundation Solver, OSi,. Then, Task B1, B2, and B3 are running solver with different method for problem 1. They are independent each other and can run in parallel. But they depends on task A, and task D depend on them. Solutions and logging information (running time) are recoded to database. After getting solution and running time from 3 methods, workflow will activate task D, call R to analyse data. R script file run analysing scenario to do experiment, find significant results as well as visualize results.

This workflow is implemented in Python, SQLite, R as details. (see Appendix)

```
[Orchestration]
database=data/py_gurobi_r.db
iterations=2
dimension=11
R=C:/Program Files/R/R-3.1.1/bin/RScript.exe
Script=R_SQLite3_Python_Gurobi.R
```

Figure 3: Configuration file

3 Data Collection

Input data for optimization problem in this project are collected from Python random generator. Number of cities that travelling sales man will reach are set in configuration (dimension parameter). Value ranges are [1, 10] for time(cost), early time is 0, and late time from [30,100]. Input data are stored in sqllite database. Result data also record in this database for analysing. Workflow will run 50 iterations (set in configuration file), and refresh dataset after each iteration.

Columns (6)

Column ID	Name	Type	Not Null	Default Value	Primary Key
0	method_id	INTEGER	0	null	0
1	run_id	INTEGER	0	null	1
2	solution	REAL	0	null	0
3	run_time	REAL	0	null	0
4	start_time	REAL	0	null	0
5	stop_time	REAL	0	null	0

	method_id	run_id	solution	run_time	start_time	stop_time
1	1	1	25	586.5175116503	1983.16537163318	1983.75188914483
2	2	2	25	578.888711504305	1984.47749294897	1985.05638166047

Figure 4: SQLite database

4 Traveling Salesman Problem with Time Windows

The travelling salesman problem (TSP) try to find the shortest possible path among a group of cities that salesman visits each city exactly once and returns to the origin city.

$$x_{ij} = \begin{cases} 1 & \textit{the path goes from city } i \textit{ to city } j \\ 0 & \textit{otherwise} \end{cases}$$

For i = 1, ..., n, let

u_i be an artificial variable (sequence number in which city i visited)

c_{ij} is the time (distance) from city i to city j

The model is formulated as following [1]:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in A} c_{ij}x_{ij} \\
\text{subject to} \quad & \sum_{(i,j)\in A} x_{ij} = 1 && \forall i \in V \\
& \sum_{(j,i)\in A} x_{ji} = 1 && \forall i \in V \\
& u_i + 1 - (n-1)(1-x_{ij}) \le u_j && \forall i,j \in V \setminus \{1\}, i \neq j \\
& x_{ij} \in \{0,1\} && \forall (i,j) \in A \\
& 1 \le u_i \le n-1 && \forall i \in V \setminus \{1\}
\end{aligned}$$

$$u_i + 1 - (n-1)(1-x_{ij}) + (n-3)x_{ji} \le u_j$$

$$\begin{aligned}
& 1 + (1-x_{1i}) + (n-3)x_{i1} \le u_i \\
& u_i \le (n-1) - (1-x_{i1}) - (n-3)x_{1i}
\end{aligned}$$

2 last contraints exclude subtours, they force $u_j \geq u_i + 1$,when $x_{ij} = 1$

Time Windows constraints:

Model 1:

```
for i in range(1,n+1):
    for j in range(2,n+1):
        if i != j:
            M = max(l[i] + c[i,j] - e[j], 0)
            model.addConstr(u[i] - u[j] + M*x[i,j] <= M-c[i,j], "MTZ(%s,%s)"%(i,j))
```

Model 2:

```
    for j in range(2,n+1):
        if i != j:
            M1 = max(l[i] + c[i,j] - e[j], 0)
            M2 = max(l[i] + min(-c[j,i], e[j]-e[i]) - e[j], 0)
            model.addConstr(u[i] + c[i,j] - M1*(1-x[i,j]) + M2*x[j,i] <= u[j], "LiftedMTZ(%s,%s)"%(i,j))

for i in range(2,n+1):
    model.addConstr(e[i] + gurobipy.quicksum(max(e[j]+c[j,i]-e[i],0) * x[j,i] for j in range(1,n+1) if i != j) \
                    <= u[i], "LiftedLB(%s)"%i)

    model.addConstr(u[i] <= l[i] - \
                    gurobipy.quicksum(max(l[i]-l[j]+c[i,j],0) * x[i,j] for j in range(2,n+1) if i != j), \
                    "LiftedUB(%s)"%i)
```

Model 3:

```
for j in range(2,n+1):
    model.addConstr(gurobipy.quicksum(u[i,j] + c[i,j]*x[i,j] for i in range(1,n+1) if i != j) -
                    gurobipy.quicksum(u[j,k] for k in range(1,n+1) if k != j) <= 0, "Relate(%s)"%j)

for i in range(1,n+1):
    for j in range(1,n+1):
        if i != j:
            model.addConstr(e[i]*x[i,j] <= u[i,j], "LB(%s,%s)"%(i,j))
            model.addConstr(u[i,j] <= l[i]*x[i,j], "UB(%s,%s)"%(i,j))
```

5 Result and Evaluation

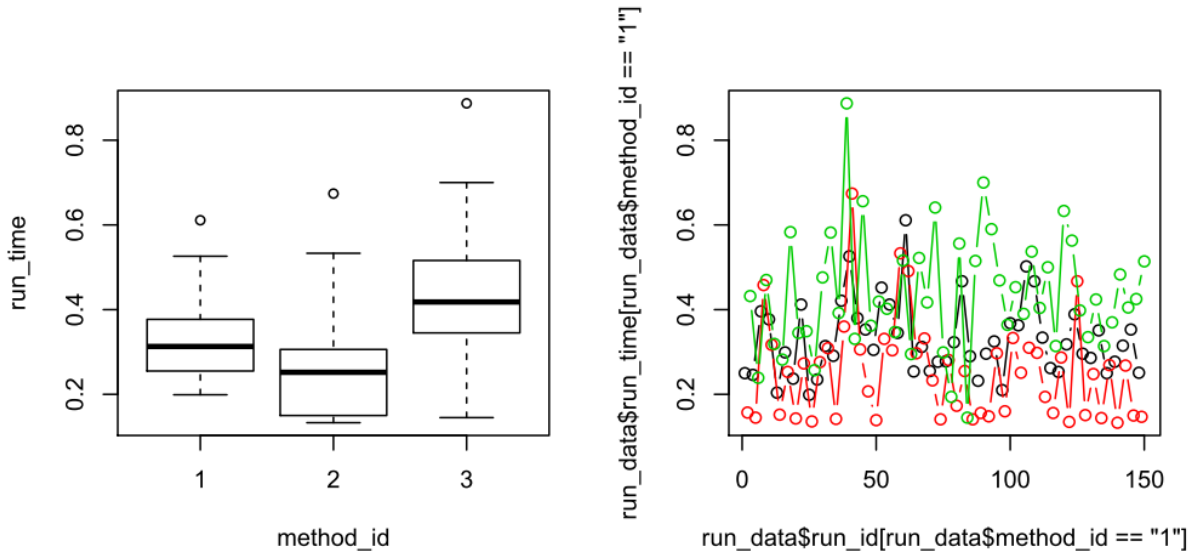

Figure 5a: Analysing result

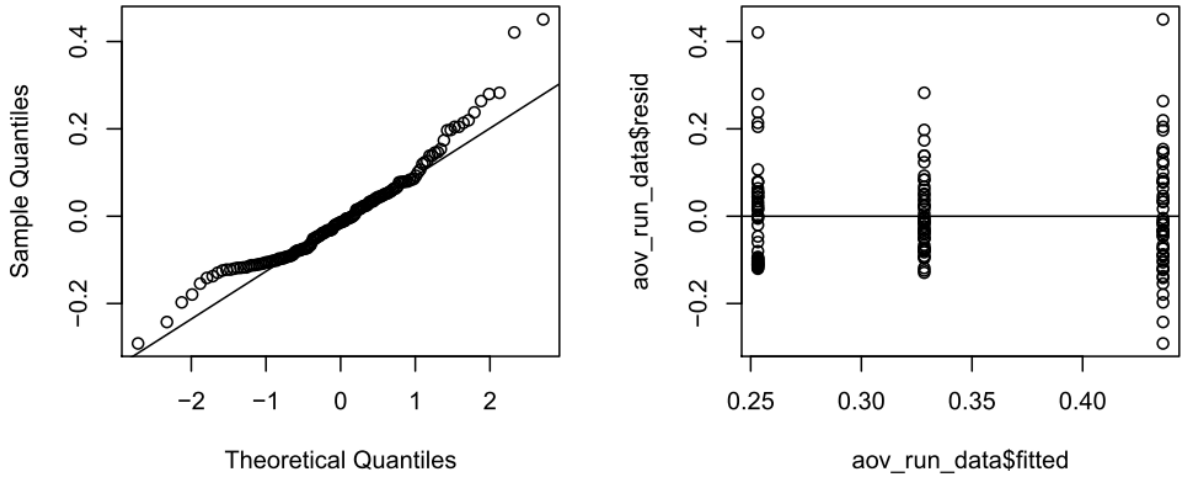

Figure 5b: QQ norm plot

```
            Df  Sum Sq Mean Sq F value Pr(>F)
method_id    2 0.06449 0.03225   1.431  0.277
Residuals   12 0.27044 0.02254
>
```

From series chart, it looks like method 2 gives smaller solving time. However ANOVA result tells us that there is no significant difference between solving time of 3 methods. Also, for small problem,

solving time is below 1s. This can be affect with large proportion of noise from computer operation. So the result is not reliable.

6 Conclusion

7 References

[1] Mikio Kubo, Joao Pedro Pedroso, Masakazu Muramatsu, Abdur Rais, *Mathematical: Solving problems using Gurobi and Python,* 2012

[2] Kate Anderson, *Managing & Sharing Your Research Data*, MU Libraries, 2014.

[3] University of Florida Data Lifecycle Management: http://ufdc.ufl.edu/IR00000801/00001

8 Appendix

File structure of workflow

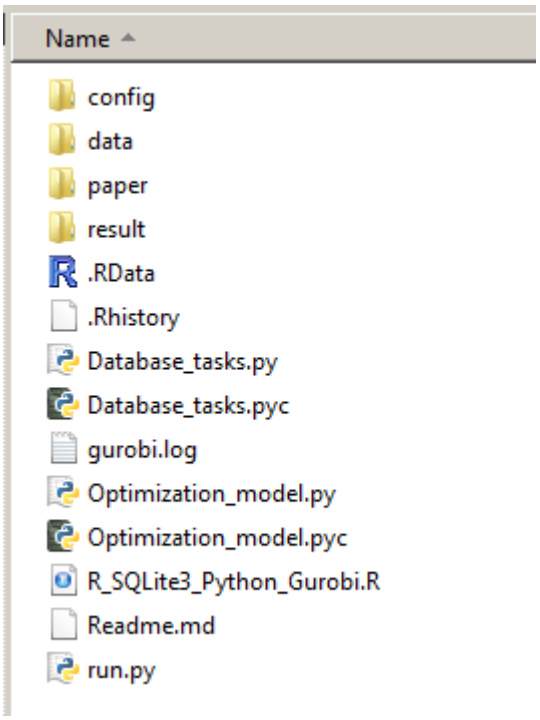

Figure 6: Project file structure

At first, we need to import following packages to Python running environment.

```
import configparser #parse configruation file
import platform

import sqlite3 as sql #work with SQLite database
import os
import random #generate data

import pyutilib.workflow #for workflow
import subprocess #run Rscript.exe
from gurobipy import * #run Gurobi solver
import timeit #record running time
```

Figure 7: Imported packages

And define workflow.

```
# @usage:
A = TaskA()
B = TaskB()
D = TaskD()
B.inputs.database_name = A.outputs.database_name
D.inputs.B_completed = B.outputs.B_completed

w = pyutilib.workflow.Workflow()
w.add(A)
w.add(B)
w.add(D)
print(w(file_name='py_gurobi_r.db'))
```

Figure 8: Workflow

Task A will open database, and configuration file.

```
class TaskA(pyutilib.workflow.Task):
    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('file_name')
        self.outputs.declare('database_name')

    def execute(self):
        #CREATE OR OPEN DATABASE
        self.database_name = generate_db()

        #READ CONFIGURATION FILE
        config = configparser.ConfigParser()
        config.read('config.ini')
        program = config['Orchestration']['R']
        argument = config['Orchestration']['Script']
```

Figure 9: Task A

Generate database and insert data.

```
def generate_db():
    db_name = 'py_gurobi_r.db'
    db = sql.connect(db_name)
    db.row_factory = sql.Row #use to access column by name (otherwise, column mus


    #CREATE TABLE
    cursor = db.cursor()

    sql_stm = 'DROP TABLE IF EXISTS TRAVEL_COST'
    cursor.execute(sql_stm)
    sql_stm = 'CREATE TABLE TRAVEL_COST(start_node INTEGER, end_node INTEGER, cos
    cursor.execute(sql_stm)

    sql_stm = 'DROP TABLE IF EXISTS TIME'
    cursor.execute(sql_stm)
    sql_stm = 'CREATE TABLE TIME(node INTEGER, early_time INTEGER, late_time INTE
    cursor.execute(sql_stm)

    sql_stm = 'DROP TABLE IF EXISTS RUNS'
    cursor.execute(sql_stm)
    sql_stm = 'CREATE TABLE RUNS(method_id INTEGER, run_id INTEGER PRIMARY KEY, s
    cursor.execute(sql_stm)

    db.commit()


    #INSERT VALUES TO TABLE
    n = 10 #number of nodes
    for i in range(1,n+1):
        for j in range(i+1,n+1):
            start_node = i
            end_node = j
            cost = random.randint(1,10)
            cursor.execute('''INSERT INTO TRAVEL_COST(start_node,end_node,cost)
                           VALUES(?,?,?)''', (start_node,end_node,cost))

    for i in range(1,n+1):
        node = i
        early_time = random.randint(0,0)
        late_time = random.randint(30,100)
        cursor.execute('''INSERT INTO TIME(node,early_time,late_time)
                       VALUES(?,?,?)''', (node,early_time,late_time))
```

Figure 10: Open SQLite database and generate data

Task B will solve problem 1

```
class TaskB(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('database_name')
        self.outputs.declare('B_completed')

    def execute(self):
        k = 50
        for i in range(1,k+1):
            n,c,e,l = read_db(refresh_db(self.database_name))
            model_1 = mtztw(n,c,e,l)
            model_2 = mtz2tw(n,c,e,l)
            model_3 = tsptw2(n,c,e,l)
            print "\n**************mtztw Method********************",i,"/",k
            print('############ TRAVELING SALESMAN PROBLEM PROBLEM WITH TIME WINDOWS
            print "**************mtztw Method********************"

            start_time = timeit.default_timer()
            model_1.optimize()
            stop_time = timeit.default_timer()
            print "START TIME = ", start_time
            print "STOP TIME = ", stop_time
            run_time = (stop_time - start_time)*1000 #mili second
            print "SOLVING TIME = ", run_time
            print "OPTIMAL VALUE =", model_1.ObjVal
```

Figure 11: Task B for problem 1

Model for task B

```
def mtztw(n,c,e,l):#############################################################
    """mtzts: model for the traveling salesman problem with time windows
    Parameters:
        - n: number of nodes
        - c[i,j]: cost for traversing arc (i,j)
        - e[i]: earliest date for visiting node i
        - l[i]: latest date for visiting node i
    Returns a model, ready to be solved.
    """

    model = gurobipy.Model("tsptw - mtz")
    x,u = {},{}
    for i in range(1,n+1):
        u[i] = model.addVar(lb=e[i], ub=l[i], vtype="C", name="u(%s)"%i)
        for j in range(1,n+1):
            if i != j:
                x[i,j] = model.addVar(vtype="B", name="x(%s,%s)"%(i,j))
    model.update()

    for i in range(1,n+1):
        model.addConstr(gurobipy.quicksum(x[i,j] for j in range(1,n+1) if j != i) == 1, "Out(%s)"%i)
        model.addConstr(gurobipy.quicksum(x[j,i] for j in range(1,n+1) if j != i) == 1, "In(%s)"%i)

    for i in range(1,n+1):
        for j in range(2,n+1):
            if i != j:
                M = max(l[i] + c[i,j] - e[j], 0)
                model.addConstr(u[i] - u[j] + M*x[i,j] <= M-c[i,j], "MTZ(%s,%s)"%(i,j))

    model.setObjective(gurobipy.quicksum(c[i,j]*x[i,j] for (i,j) in x),gurobipy.GRB.MINIMIZE)

    model.update()
    model.__data = x,u
    return model
```

Figure 12: Model of task B

Task D will call R to analyse and visualize results.

```
class TaskD(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('B_completed')
        self.outputs.declare('result_D')

    def execute(self):
        if (self.B_completed == True):
            config = configparser.ConfigParser()
            config.read('config.ini')

            program = config['Orchestration']['R']
            argument = config['Orchestration']['Script']

            subprocess.call([program, argument])
            print "\n**********************************"
            print('############# FINISH RUN R ################')
            print "**********************************"

            self.result_D = "R Scripts Succeeded"
        else:
            self.result_D = "Cannot receive result from B to continue"
```

Figure 13: Task D will run R script

R Script

```r
library("DBI")
library("RSQLite")

drv <- dbDriver("SQLite")
con <- dbConnect(drv, "py_gurobi_r.db")
dbListTables(con)
dbListFields(con, "RUNS")
run_data = dbGetQuery(con, "Select method_id, run_id, solution, run_time, start_time, stop_time fr
length(run_data)

run_data$method_id = as.factor(run_data$method_id)
str(run_data)
aov_run_data = aov(run_time ~ method_id, run_data)
sm = summary(aov_run_data);sm

pdf("plots_py_gurobi_r.pdf")
par(mfrow=c(2,2))
plt0 = plot(run_time ~ method_id,run_data)

#type "b" => show points and lines
plt1 = plot(run_data$run_id[run_data$method_id=="1"], run_data$run_time[run_data$method_id=="1"], 1
lines(run_data$run_id[run_data$method_id=="2"], run_data$run_time[run_data$method_id=="2"], col=2,
lines(run_data$run_id[run_data$method_id=="3"], run_data$run_time[run_data$method_id=="3"], col=3,

plt2 = qqnorm(aov_run_data$resid)
plt3 = qqline(aov_run_data$resid)
plt4 = plot(aov_run_data$fitted,aov_run_data$resid)
plt5 = abline(h=0)  #Add the horizontal axis
dev.off()
```

Figure 14: R script for analysing data