# Solutions of Quiz

Arezoo Jahani

(a.jahani@tabrizu.ac.ir)

May 24, 2020

Optimization is the way of life. We all have finite resources and time and we want to make the most of them.

**Challenge 1** Describe one method/algorithm that is commonly used to solve the following linear program:

$$\begin{aligned} max_x \quad & c^T X \\ s.t: Ax &\leq b, \\ x &\geq 0, \end{aligned} \tag{1}$$

where $x \in R^n$ represents the vector of $n$ unknown variables, $c \in R^n$ is a vector of known coefficients, $b \in R^m$ is a vector of given constraint values, and $A \in R^{m \times n}$ is a matrix with known coefficients.

Linear programming (LP) is one of the simplest ways to perform optimization. It helps us solve some very complex optimization problems by making a few simplifying assumptions. Linear programming is a simple technique where we depict complex relationships through linear functions and then find the optimum points. There are some solutions for solving linear programming like:

- Solve Linear Program by Graphical Method: this method is used to solve problems by finding the highest or lowest point of intersection between the objective function line and the feasible region on a graph.
- Solve Linear Program using R: R is an open-source tool for LP solving which is popular in data scientists experts. It has some open source packages like "lpSolve".
- Solve Linear Program using OpenSolver: OpenSolver is an open-source linear and optimizer for Microsoft Excel. It is an advanced version of a built-in Excel Solver.
- Simplex Method: The simplex method is an iterative procedure for getting the most feasible solution. In this method, we keep transforming the value of basic variables to get maximum value for the objective function.

- Northwest Corner Method and Least Cost Method: The northwest corner method is used to calculate the feasible solution for transporting commodities from one place to another. Least Cost method is another method to calculate the most feasible solution for a linear programming problem. This method derives more accurate results than Northwest corner method. It is used for transportation and manufacturing problems.

Here I want to use open-source packages and free solvers in Julia programming language. The JuMP package is an open-source optimization package and solvers like "CPLEX", "Cbc", "GLPK" and "Gurobi" can be used in this language. Although Gurobi is not open source, we can use it under an educational license. The other solvers are open source and easy to use.

**Challenge 2** Implement a function in the programming language Julia (www.julialang.org) that solves the problem given in (1). Note: Use an open-source solver and use existing Julia packages.

In order to solve the linear programming problem, the JuMP package (Modeling language for Mathematical Optimization) in the programming language Julia is used. Also the "GLPK" solver as an open-source solver is used in this section. It should be noted that in this section, we have quantified the variables and parameters of the problem by default (*main.jl*). But in the next challenge, it is possible to change these values and generate (*Generate_data.jl*) and execute (*LP.jl*) larger problems.

The following program shows the contents of the file (*main.jl*), which is a function of solving the problem of optimization of challenge 1 with the default parameters and variables.

```
// main.jl
using JuMP, GLPK
using DelimitedFiles
m = Model(GLPK.Optimizer)

# Initialization  the parameters
c= [1 2 3]
A= [9 8 7;6 5 4]
b = [10;11]

@variable(m, x[1:size(c,2)] >= 0 )
@objective(m, Max, sum(c*x) )
@constraint(m, con, A*(x) .<= b )
print(m)

status  = optimize!(m)

println("Objective value: ", JuMP.objective_value(m))
println("x[1]  = ", JuMP.value(x[1]))
println("x[2]  = ", JuMP.value(x[2]))
```

**Challenge 3** Give a numerical solution to a problem of your choice with $n = 3$ and $m = 2$.

The following results show the results of running the file (*main.jl*) with dimension $n = 3$ and $m = 2$ and randomly generated parameters.

```
// main.jl 2 3 (a numerical solution of problem with n = 3 and m = 2)
Max 72 x[1] + 50 x[2] + 46 x[3]
Subject to
 99 x[1] + 46 x[2] + 27 x[3] <= 16.0
 11 x[1] + 9 x[2] + 84 x[3] <= 29.0
 x[1] >= 0.0
 x[2] >= 0.0
 x[3] >= 0.0
Objective value: 22.863849765258216
x[1] = 0.0
x[2] = 0.15492957746478878
```

**Challenge 4** Describe the changes that would need to be made to your program for an arbitrarily large matrix A, i.e., $n \gg 1$ and $m \gg 1$ (500 words max).

In order to have a solution that is used for any inputs with different dimensions, I wrote another function (*Generate_data.jl*), only for generating random data by receiving the required dimension. For example it can be run by *cmd* and writing the name of the file and two inputs respectively show $m$ and $n$. How to use this function is described below:

```
// Generating data with required dimensions of m and n
> julia Generate_data.jl m n
such as:
> julia Generate_data.jl 2 3
```

After generating required data with required dimensions, now we can solve the problem with generated data. In order do this work, we need to run (*LP.jl*) function again with two inputs. The inputs show the dimensions $m$ and $n$ respectively. Adding these inputs in the running process, force the code to use the generated data in the previous section and solve the problem constructed with those generated data. How to use this function is described below:

```
// Solving the linear problem with generated data with required dimensions
> julia LP.jl m n
such as:
> julia LP.jl 2 3
```

In order to solve the problem with large matrix $A$, we can use the options like $MIPGap$. The solver will terminate (with an optimal result) when the gap between the lower and upper objective bound is less than MIPGap times the absolute value of the upper bound. It only affects mixed integer programming (MIP) models.

**Appendix**

This section first explains how to generate data with different dimensions, which is implemented in file ($Generate_data.jl$). It then describes how to solve the optimization problem created with the generated data, which is implemented in file (*LP.jl*).

As shown in the file ($Generate_data.jl$), During the running, the program receives two numbers as input which indicates the dimensions of the problem and generates required parameters randomly in a folder called *data*. Then, file (*LP.jl*), by receiving two inputs as the number of dimensions of the problem, reads the generated data from the *data* file and creates and solves an optimization problem with them. Then, in the end, the results of the solution are shown in the form of the value of the variables. The contents of files ($Generate_data.jl$) and (*LP.jl*) are shown below.

```
// Generate_data.jl
function main(args)
    if 2 > length(args) > 2
      println("Error! You entered more than two inputs.")
    else
      m = parse(Int,args[1])
      n = parse(Int,args[2])
      path = "data/" * string(m) * "-" * string(n);
      try
        mkdir(path)
      catch
      end

      # Generating Parameter c
      io = open(path * "/c.txt", "w")
      result = rand(collect(1:100),1,n)
      for i=1:length(result)
        print(io, result[i])
        if i!= length(result)
          print(io, " ")
        end
      end
      close(io)

      # Generating Parameter b
      io = open(path * "/b.txt", "w")
      result = rand(collect(1:100),m,1)
      for i=1:length(result)
        print(io, result[i])
        if i!= length(result)
          write(io, "\n")
        end
      end
      close(io)

    # Generating Parameter A
    io = open(path * "/A.txt", "w")
    result = rand(collect(1:100),m,n)
    for i=1:size(result,1)
      for j=1:size(result,2)
        print(io, result[i,j])
        if j!= size(result,2)
```

```
          write(io, " ")
        end
      end
      if i!= size(result,1)
        write(io, "\n")
      end
    end
    close(io)
    end
end
main(ARGS)
```

```
// LP.jl
using JuMP, GLPK
using DelimitedFiles

function main(args)
  if 2 > length(args) > 2
    println("Error! You entered more than two inputs.")
  else
    m = parse(Int,args[1])
    n = parse(Int,args[2])
    model = Model(GLPK.Optimizer)

    # reading parameters
    c = readdlm("data/" * string(m) * "-" * string(n) * "/c.txt")
    A = readdlm("data/" * string(m) * "-" * string(n) * "/A.txt")
    b = readdlm("data/" * string(m) * "-" * string(n) * "/b.txt")

    @variable(model, x[1:size(c,2)] >= 0 )
    @objective(model, Max, sum(c*x) )
    @constraint(model, con, A*(x) .<= b )
    print(model)

    status = optimize!(model)

    println("Objective value: ", JuMP.objective_value(model))
    println("x[1] = ", JuMP.value(x[1]))
    println("x[2] = ", JuMP.value(x[2]))
  end
end

main(ARGS)
```