Final Project
Linear & Integer Programming

CS110 - Professor Stern, P.

Ash Nguyen

2018/12/06

1 Linear Programming

Linear programming is a method of optimization that aims to solve problems of maximizing or minimizing a linear function of multiple variables, with linear constraints. There are several methods to solve linear programs, notably two class of algorithmic solver: path-walking algorithms, such as the simplex algorithm, or interior point algorithm, such as the ellipsoid algorithm. In this section, I'll discuss and implement the simplex algorithm to solve linear programs.

1.1 Python Implementation: Simplex Method

The simplex method constitute of two phase implementation. In phase I, the problem is checked for feasibility, and slack variables are added if the original linear program requires slack variables. In phase II, the linear program is rewritten in canonical form, then there are multiple methods to perform path-walking to reach the optimal solution, assuming the original linear program behaves nicely (no cycling). In this implementation, I merge the two phase to add slack variables without checking for theirs necessities. I also do not introduce feasibility check, and my alternative is to perform path-walking for a definite number of time (10000 times), after which infeasibility will be announced.

A standard linear program is presented in the form:

Maximize:

$$a_1x_1 + +a_2x_2 + a_3x_3 + ... + a_nx_n$$

subject to:

$$\begin{aligned} b_{10}x_1 + b_{20}x_2 + b_{30}x_3 + ... + b_{n0}x_n &\leq 0 \\ b_{11}x_1 + b_{21}x_2 + b_{31}x_3 + ... + b_{n1}x_n &\leq 0 \\ &\vdots \\ b_{1m}x_1 + b_{2m}x_2 + b_{3m}x_3 + ... + b_{nm}x_n &\leq 0 \\ x_1, x_2, x_3, ..., x_n &\geq 0 \end{aligned}$$

The linear constraints create a feasible region of a polytope, restricting the solution by m number of half-spaces. The geometric interpretation of the simplex algorithm is walking through the edges of a high-dimensional polytope until the maximum point is reached. In this implementation, I used pivoting for each iteration to reach the optimal solution. The procedure is of following:

1. Constructing the canonical table from the input.

2. Choose pivot element by the following criteria: the column is the most negative element of the objective function and the row is the least nonnegative (to prevent cycling) ratio between the right hand side of the constraint matrix and the element at the chosen column. This choice of pivot element ensures the iterated pivot table will always produce a *better* result for the objective function (not necessarily the optimal choice of pivot element).

3. Perform Gaussian elimination on the chosen pivot column to have the reduced echelon form of the matrix.

4. Check the optimality condition: if the entry in the objective function row of the pivot table is all nonnegative, the optimal solution is reached and the algorithm terminates.

5. Otherwise, go back to step 2.

The following code implement the simplex algorithm described above:

```
'''
Python Implementation Of Simplex Method For
Solving Linear Program In Canonical Formulation
'''

#importing relevant libraries
import numpy as np
import math

#Function to choose pivot column, which is the most negative entry of the
#objective row (last row in this implementation)
```

```
12 def choose_col(table):
13     low = 0
14     index = 0
15     for i in range(1, table.shape[0]-1):
16         if table[table.shape[0]-1,i] < low:
17             low = table[table.shape[0]-1,i]
18             index = i
19     return index
20 
21 #Function to choose pivot row, which is the least nonnegative entry of the
       ratio
22 #between the cRHS and the entry itself, in the chosen pivot column
23 def choose_row(table, col):
24     low = math.inf
25     index=-math.inf
26     for i in range(0,table.shape[0]-1):
27         if low>table[i,table.shape[1]-1]/table[i,col] and table[i,table.
       shape[1]-1]/table[i,col]>=0:
28             low=table[i,table.shape[1]-1]/table[i,col]
29             index=i
30     return index
31 
32 #Fucntion to pivot the chosen entry by Gaussian-Jordan elimination between
       this row and the other
33 #By construction of the simplex algorithm (the chosen row and column), this
        elimination will always
34 #lead to an improved objective function
35 def pivot(table,step):
36     print("At step", step+1)
37     print("___Before Pivot:\n", table)
38     #retrieve the chosen row and column
39     col=choose_col(table)
40     row=choose_row(table,col)
41     #Substracting the normalized chosen row from other row to have the
42     #reduced row echelon form matrix, which is our new table to pivot
43     #(or the optimal result)
44     for i in range(table.shape[0]):
45         if i != row:
46             ratio=table[i,col]/table[row,col]
47             table[i,:]=table[i,:]-table[row,:]*ratio
48     print("___After Pivot:\n", table)
49     return table
50 
51 #Function to check the optimality of the algorithm: if optimality
52 #is reached, all entries of the objective row will be nonnegative
53 def optcheck(tab):
54     dummy=np.array(tab[tab.shape[0]-1,])
55     if np.all(dummy>=0):
56         return True
57     else: return False
58 
59 #Main function for simplex algorithm
60 def simplex(obj,cLHS,cRHS):
61     #checking the input form of the linear program to see if
```

```
    #it fits the allowed type of input
    test=np.matrix([0])
    assert type(obj)==type(test)
    assert type(cLHS)==type(test)
    assert type(cRHS)==type(test)
    #Constructing the canonical simplex matrix
    obj=np.concatenate((-obj,np.zeros((1,cLHS.shape[0]+1))),axis=1)
    #Adding slack variable, 1 slack for each constraints,
    #no surplus variable since we assume the simplest canonical form
    slack=np.identity(cLHS.shape[0],dtype=float)
    cLHS=np.concatenate((cLHS,slack),axis=1)
    constraints=np.concatenate((cLHS,cRHS),axis=1)
    tab=np.concatenate((constraints,obj),axis=0)
    #Main loop to find the optimal solution
    i=0
    while optcheck(tab)==False and i<=10000:
        tab=pivot(tab,i)
        i+=1
    if i==10000:
        print('Infeasible')
    else:
        print('The optimal value of the objective function is:', tab[tab.
    shape[0]-1,tab.shape[1]-1])

'''
Solving a linear program in canonical form:
    Maximizing: 2a-b+2c
    Subject to: 2a+b<=10 ; a+2b-2c<=20; b+2b<=5
'''

A=np.matrix([2,-1,2])
B=np.matrix([[2,1,0],[1,2,-2],[0,1,2]])
C=np.matrix([[10],[20],[5]])
simplex(A,B,C)
```

If we have n variables and m constraints, the complexity is $O(m+n)$, since at most we run a for loop in the pivoting process, and the main while loop only run a definite number of time before it terminates. The space complexity for the solver itself is constant since we only store the pivot table when we are pivoting it.

The algorithm produces the result:

```
At step 1
---Before Pivot:
 [[  2.   1.   0.   1.   0.   0.  10.]
 [  1.   2.  -2.   0.   1.   0.  20.]
 [  0.   1.   2.   0.   0.   1.   5.]
 [ -2.   1.  -2.   0.   0.   0.   0.]]
---After Pivot:
 [[  2.   1.   0.   1.   0.   0.  10.]
 [  1.   3.   0.   0.   1.   1.  25.]
 [  0.   1.   2.   0.   0.   1.   5.]
```

```
 [ -2.   2.   0.   0.   0.   1.   5.]]
At step 2
---Before Pivot:
 [[  2.   1.   0.   1.   0.   0.  10.]
 [  1.   3.   0.   0.   1.   1.  25.]
 [  0.   1.   2.   0.   0.   1.   5.]
 [ -2.   2.   0.   0.   0.   1.   5.]]
---After Pivot:
 [[  2.    1.    0.    1.    0.    0.   10. ]
 [  0.    2.5   0.   -0.5   1.    1.   20. ]
 [  0.    1.    2.    0.    0.    1.    5. ]
 [  0.    3.    0.    1.    0.    1.   15. ]]
The optimal value of the objective function is: 15.0
```

The implemented linear program solver is quite limited in its capacity to spot infeasibility and the bounds of the variables themselves. Because of these limitations, even though the solver has linear time complexity, it perform quite bad for even small problem of 50 variables generated randomly. In terms of capabilities, there are two Python packages that implement linear programs solver in a more effective manner: scipy.optimize and CVXPY.

1.2 Comparison With scipy.optimize & CVXPY

Scipy.optimize also uses simplex algorithm to solve linear programs, but it does so with a full two phase simplex method implementation, thus allows feasibility to be spotted. CVXPY on the other hand uses a built-in convex optimizer (in this case CVXOPT) to solve the linear program as a convex program using the KKT condition, which in practice is much more efficient than the simplex method. This efficient is due to the fact that differentiation of a linear object and linear constraints are relatively simple and low-cost compared to repeatedly pivoting elements in a dynamic matrix. A practical comparison between these solvers for linear program demonstrates this speculation.

```
'''
Comparing Linear Program Solvers:
- My Implementation Of Simplex Method
- scipy.optimize.linprog Simplex Method
- CVXPY various optimizers
'''

#Importing relevant libraries
import timeit
from scipy.optimize import linprog as lp
import cvxpy as cvx
import random
import numpy as np
import matplotlib.pyplot as plt


#Wrapper functions to solve arbitrary large linear programs using
```

```
#randomly generated coefficients
def linprog_simplex(N,M):
    A=np.matrix([random.randint(-100,100) for _ in range(N)])
    B=np.matrix([[random.randint(-100,100) for _ in range(N)] for _ in
    range(M)])
    C=np.matrix([[random.randint(-12500,25100)] for _ in range(M)])
    simplex(A,B,C)

def linprog_scipy(N,M):
    A=[random.randint(-100,100) for _ in range(N)]
    B=[[random.randint(-100,100) for _ in range(N)] for _ in range(M)]
    C=[[[random.randint(-12500,25100)] for _ in range(M)]]
    lp(A,A_ub=B,b_ub=C,method='simplex')

def linprog_cvx(N,M):
    A=np.matrix([random.randint(-100,100) for _ in range(N)])
    B=np.matrix([[random.randint(-100,100) for _ in range(N)] for _ in
    range(M)])
    C=np.matrix([[random.randint(-12500,25100)] for _ in range(M)])
    x=cvx.Variable(N,1)
    obj=cvx.Minimize(A*x)
    constraint=[B*x<=C,x>=0]
    prob=cvx.Problem(obj,constraint)
    prob.solve()

#Wrapper function for timeit module
def wrap(func, args1,args2):
    def wrapped():
        return func(args1,args2)
    return wrapped


dummy=[]
scipy=[]
cvxpy=[]

for i in range(1,400):
    dummy.append(i)
    scipy.append(timeit.timeit(wrap(linprog_scipy,i,i),number=100)/100)
    cvxpy.append(timeit.timeit(wrap(linprog_cvx,i,i),number=100)/100)


plt.plot(dummy,scipy,color='red',label='scipy')
plt.plot(dummy,cvxpy,color='green',label='cvxpy')
plt.legend()
plt.xlabel('Number of variable and constraints')
plt.ylabel('Time')
plt.show()
```

As we can see in figure 1, scipy.optimize starts off as good as CVXPY for small problems (¡80 variables and ¡80 constraints), but then perform quite worse because of the pivoting process. It is understandable though, since the simplex method is quite numerical, and even though it guarantees convergence toward the optimal solution it does

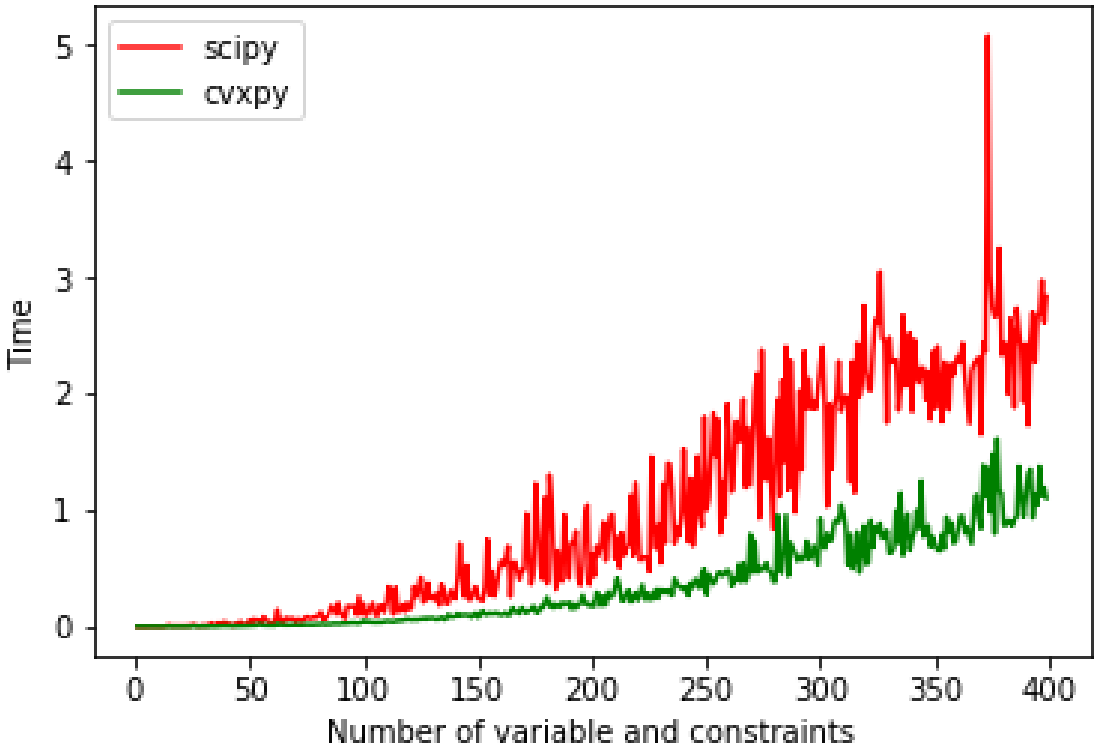

Figure 1: Practical runtime comparison between scipy.optimize & CVXPY for LPs

not pick the optimal path at every iteration, resulting in inefficient walking around the edges of the polytope while CVXOPT is implemented with heuristic line search for every iteration.

2 Application: The Traveling Salesman Problem

2.1 Introduction

An extension of the linear program is the integer program, where all problem formulations are similar with an added constraint that all variables are integers. With proper formulation, integer programs can be used to solve very hard problem, such as the Traveling Salesman Problem (TSP).

The TSP is an NP-complete problem, so no efficient exact algorithm currently exists. The problem is to find a closed path in a directed graph so that every vertex in the graph is visited exactly once. With a brute-force approach, TSP requires a complexity of $O(n!)$ for n vertices because it needs to try every possible route between every vertices, making

it impractical for even small number of vertex.

2.2 Dynamic Programming Approach

Since I need to construct a path that visits every vertices once and only once, I can start with an arbitrary vertex, which I will now call vertex 1. The minimum distance path can be expressed as:

$$path_{min} = min\left(d[i->1] + path_{min_{1->i}}\right) \qquad i \in \{1, 2, ..., n\}$$

where $d[i->1]$ is the distance between vertex i and vertex 1; $path_{min_{1->i}}$ is the smallest path connecting 1 and i that visits all other vertices.

Now, the problem exhibits optimal substructure: the smallest path connecting every vertices in a set S that starts from 1 and ends at i is optimal only if all of its segments connecting the vertices are optimal. Thus, we can do a bottom-up dynamic program that record all possible such smallest path for every permutation of size 2,3... and n vertices. Let S_k be the set of all possible k vertices permutations that contains vertex 1, we have the following recursive relation:

$$min_{S_k,j} = min\left(min_{S_k - i,j} + d[i->j]\right) \qquad j \in S_k; j \neq i; j \neq 1$$

The base case will be $S_2 = 1, i$, and the distance minimum distance is simply the relation we have above: $min(d[i->1])$. Because it is a bottom-up dynamic program, we need to compute the answers to all possible set, with all possible combination of vertex (i.e. all possible $k \in \{2, 3, ..., n\}$ and $i \in \{1, 2, ..., n\}$). There are 2^n possible such choices of all permutations of S_k for all k, therefore the time complexity for a dynamic program with bottom up approach is $O(2^n n)$, which is better than $O(n!)$ but still an exponential time. The space complexity for the algorithm is also $O(2^n n)$, since we have to keep track of all possible permutation paths. We can reduce the space complexity into $O(n^3)$ if we can have a three dimensional tensor.

2.3 Integer Programming Approach

The TSP can be formulated as an integer program. Let x_{ij} be a binary decision variable that represents whether we choose to include the path from vertex i to vertex j in our optimal path or not, and let d_{ij} be the distance between vertex i and vertex j, we have the following integer program:

Minimize:

$$O = \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij}$$

subject to:

$$\sum_{i=1}^{n} x_{ij} = 1 \tag{1}$$

$$\sum_{j=1}^{n} x_{ij} = 1 \tag{2}$$

$$x_{ij} \in \{0, 1\} \tag{3}$$

where condition (1) and (2) guarantees that a vertex has exactly one degree in and one degree out, making sure we have a closed path. This formulation is valid, and it works in some cases, but it does not prevent subtours (i.e. instead of $1- > 2- > 3- > 4- > 5- > 6- > 1$ the algorithm returns $1- > 2- > 3- > 1$ and $4- > 5- > 6- > 4$, which also satisfy the one-in one-out condition). To prevent subtours, I use the MTZ condition:

$$u_i - u_j + nx_{ij} \leq n - 1 \quad i \neq j; \;\; i, j = 2..., n$$

where u_i is a nonnegative auxiliary variable.

Implementation of this integer program for CVXPY is shown below. CVXPY is choses because its effective branch-and-bound method of solving integer programs. Scipy.optimize can be used to build a branch-and-bound solver for integer programs, but the linear program solvers of scipy.optimize itself lacks the ability to return exact results (for example, integer solution of 2 can be returned as 1.9999999) and therefore will make a branch-and-bound implementation unbounded. CVXPY, on the other hand, has a readily built-in class of integer variable that self-restricted itself to integer values (up to a certain degree of floating point). It also has a Boolean variable class built-in that can be used conveniently as binary variable.

```
'''
Solving The Traveling Salesman Problem Using
Integer Programming
'''
#import relevant libraries
import cvxpy as cvx
import numpy as np
import math
import timeit
import matplotlib.pyplot as plt

#main tsp solver that accepts a matrix distance A as the input
def tsp_integer(A):
    N=A.shape[0]
    #A Boolean matrix that represents the choice of path
    trip=cvx.Bool(N,N)
    #A column vector to specify constraint (1) and (2)
    aux=np.ones((1,N))
```

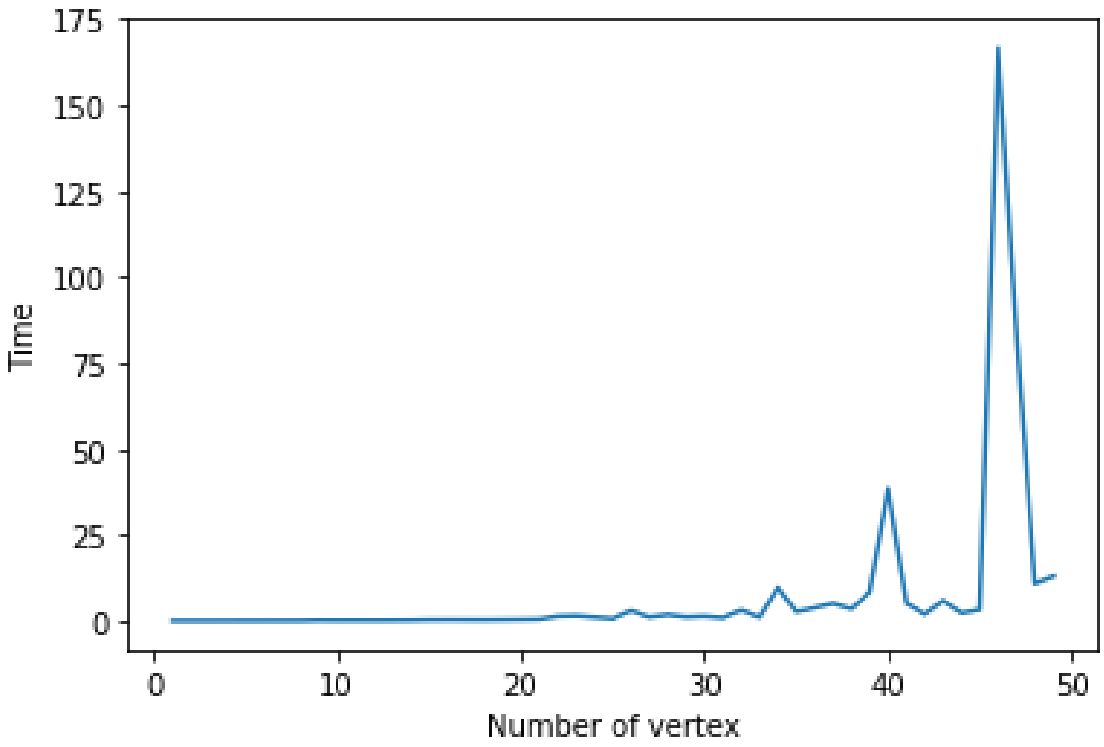

Figure 2: Practical runtime for TSP solution with CVXPY IP solver

```
    #An auxiliary nonnegative variable to specify the MTZ condition
    mtz=cvx.Variable(N,1)
    for i in range(N):
        constraints.append(mtz[i] >= 0)
    #Objective: minimize the strip, subject to the choice of the Boolean
    variables
    obj = cvx.Minimize(sum([A[i,:]*trip[:,i] for i in range(N)]))
    #One-in one-out degree constraints
    constraints = [(cvx.sum_entries(trip, axis=0) == aux),
                   (cvx.sum_entries(trip, axis=1) == aux.transpose())]
    #MTZ constraints
    for i in range(1,N):
        for j in range(1,N):
            if i != j:
                constraints.append(mtz[i] - mtz[j] + N*trip[i,j] <= N-1)
    #Solve using CVXPY
    problem=cvx.Problem(obj,constraints)
    problem.solve()
    return problem.value

def wrap(func, args):
    def wrapped():
        return func(args)
```

```
41      return wrapped
42
43 dummy=[]
44 time=[]
45 for i in range(1,50):
46      time.append(timeit.timeit(wrap(tsp_integer,np.identity(i)*100000+np.
        random.rand(i,i)),number=100)/100)
47      dummy.append(i)
48
49 plt.plot(dummy,time)
50 plt.xlabel('Number of vertex')
51 plt.ylabel('Time')
52 plt.show()
```

The MTZ constraint uses two for loops, so the dominate complexity of the problem formulation part is $O(n^2)$. However, depends on the nature of the distance matrix, branch-and-bound method can take anywhere from 1 to 2^n branches to solve the integer program. Therefore, this method is proven empirically to be effective for only around 200 vertices.

References

[1] Wikipedia. (n.d.). *Travelling salesman problem*. Retrieved from:
https://en.wikipedia.org/wiki/Travelling_salesman_problem

[2] Karp, R., Held, M. (2006). *A Dynamic Programming Approach to Sequencing Problems*. Retrieved from:
https://epubs.siam.org/doi/10.1137/0110015

[3] CVXPY Documentation (n.d.). *CVXPY Documentation: Advanced Features*. Retrieved from:
http://www.cvxpy.org/en/latest/tutorial/advanced/index.html

[4] Scipy Source Code. (n.d.). */Scipy/scipy/optimize/linprog.py*. Retrieved from:
https://github.com/scipy/scipy/blob/master/scipy/optimize/_linprog.py

[5] Taylor, B.W. (n.d.).*Introduction to Management Science: Integer Programming - The Branch and Bound Method*. Retrieved from:
https://media.pearsoncmg.com/ph/bp/bridgepages/bp_taylor_bridgepage/taylor_13e/online_modules/Taylor13_Mod_C.pdf