CS 3430: S24: Scientific Computing
Assignment 04
Simplex Algorithm

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 3, 2024

Learning Objectives

1. Linear Programming
2. Standard Maximum Problems
3. Tableaus and Tableau Formation
4. Entering and Departing Variables
5. Pivots and Pivoting Operation
6. Simplex Algorithm

Introduction

In this assignment, we will build up on what we learned in Assigned 03, where we solved linear programming (LP) problems in a semi-automated fashion by graphing them, using the graph to compute the corner points, and then maximizing/minimizing the object function at the corner points. The simplex algorithm allows us to solve standard maximum problems (SMPs) in any number of dimensions with pure algebra and no graphs. You will save your coding solutions in `cs3430_s24_hw04.py` included in the zip and submit it in Canvas.

Problem 0: (0 points)

Review the PDFs of Lectures 06 and 07 in Canvas and/or your lecture notes. Make sure that you are intellectually comfortable with such concepts as *slack variables*, *slack equations*, *basic solutions*, *tableaus*, *tableau formation*, *pivots*, *pivoting operation*, *entering variable*, *departing variable*, and *simplex algorithm*. In reviewing the simplex algorithm (slides 32 – 37 in the Lecture 07 PDF) make sure you understand the two stopping conditions of the simplex algorithm.

Coding Lab 1: (0 points)

Let us consider the following optimization problem.

```
A fertilizer company makes the following fertilizer types:
20-8-8 for lawns, 4-8-4 for gardens, and 4-4-2 for general purposes.
The numbers in each fertilizer brand refer to the weight percentages
```

of nitrate, phosphate, and potash, respectively, in one fertilizer sack. The company has 6,000 pounds of nitrate, 10,000 of phosphate, and 4,000 pounds of potash. The profit is $3 per 100 pounds of lawn fertilizer, $8 per 100 pounds of garden fertilizer, and $6 per 100 pounds of general purpose fertilizer. How many pounds of each fertilizer should the company produce to maximize the profit and satisfy the constraints?

We start with the decision variables: x – the number of 100's (to minimize the numbers of 0's we have to deal with) of pounds of 20-8-8 fertilizer; y – the number of 100's of pounds of 4-8-4 fertilizer; and z – the number of 100's of pounds of 4-4-2 fertilizer. Using these decision variables, we can express the constraints on the amounts of nitrate, phosphate, and potash as

1. $20x + 4y + 4z \leq 6,000$ (nitrate);
2. $8x + 8y + 4z \leq 10,000$ (phosphate);
3. $8x + 4y + 2z \leq 4,000$ (potash);
4. $x \geq 0$;
5. $y \geq 0$;
6. $z \geq 0$.

The objective function to maximize is $p = 3x + 8y + 6z$. Looking at the constraints and the optimization function we can quickly verify that this is, indeed, an SMP (cf. Slide 12 in the Lecture 06 PDF). Thus, the simplex algorithm is appropriate.

We start the tableau formation for the simplex algorithm with the slack equations. We have three $\leq$ constraints. So, we add three non-negative slack variables, say, u, v, w, for the unused numbers of 100's of pounds of nitrate, phosphate, and potash, respectively, to the these constraints, thus turning the latter into the following slack equations.

1. $20x + 4y + 4z + u = 6,000$ (nitrate);
2. $8x + 8y + 4z + v = 10,000$ (phosphate);
3. $8x + 4y + 2z + w = 4,000$ (potash).

We have everything in place to form the initial tableau. Remember to multiply all coefficients of the object function, i.e., 3, 8, 6, by -1.

	x	y	z	u	v	w	B.S.
u	20	4	4	1	0	0	6,000
v	8	8	4	0	1	0	10,000
w	8	4	2	0	0	1	4,000
p	-3	-8	-6	0	0	0	0

So far so good! This is similar to the tableaus we worked out on the board in Lectures 06 and 07. However, since our objective in this assignment is to write a program to do simplex, we will benefit from simplifying the tableau setup's notation by using only the variable name x with subscripts. This is a trick in many branches of mathematics that ensures that we will never run out of variable names: since there are infinitely many natural numbers, we have x_0, x_1, x_2, ..., $x_{1300000013}$, and on, and on, and on. For this problem, we let $x = x_0$, $y = x_1$, $z = x_2$, $u = x_3$, $v = x_4$, and $w = x_5$. Then the tableau looks as follows.

	x_0	x_1	x_2	x_3	x_4	x_5	B.S.
x_3	20	4	4	1	0	0	6,000
x_4	8	8	4	0	1	0	10,000
x_5	8	4	2	0	0	1	4,000
p	-3	-8	-6	0	0	0	0

Since we have only one variable name now (i.e., x), we can simplify the tableau's notation further by dropping the x's altogether and using only the appropriate subscripts. Let us do it!

	0	1	2	3	4	5	B.S.
3	20	4	4	1	0	0	6,000
4	8	8	4	0	1	0	10,000
5	8	4	2	0	0	1	4,000
p	-3	-8	-6	0	0	0	0

The only thing that we need to remember as interpreters of this notation is 0 stands for x_0, 1 for x_1, etc. Note also that the first (unlabeled) column in the tableau contains the *entered* variables (also known as the *in-variables* or the *row variables*). These are the *current* variables that define the *current* basic solution. I italicized the word "current," because the entered/row variables vary from iteration to iteration. It is only the row variables that matter in that if a variable is not a row variable, its value in the current basic solution is assumed to be 0.

We should note that the entered variables in the initial tableau are the slack variables. The last column of the tableau contains the basic solution values and the bottom row of the tableau is the p-row. I should mention in passing that we can also replace the last column's label `B.S.` with 6, because we know that the last column is the basic solution column. But, let us keep it for the sake of clarity.

Let us pause and ask ourselves, what does the initial tableau tell us? This tableau offers the following basic solution: $x_3 = 6,000$, $x_4 = 10,000$, $x_5 = 4,000$, $x_0 = x_1 = x_2 = 0$. Remember that x_0 is the original x (i.e., x – the number of 100's of pounds of 20-8-8 fertilizer), x_1 is the original y (i.e., the number of 100's of pounds of 4-8-4 fertilizer), and x_2 is the original z (i.e., the number of 100's of pounds of 4-4-2 fertilizer). Also, recall the convention is that if the variable is not an in-variable/row variable/entered variable, its value is 0. Thus, since x_0, x_1, x_2 are the departed variables (i.e., they are not *currently* in the rows), they are equal to 0, by this convention. So, the first basic solution given by the unitial tableau is $x_0 = 0$, $x_1 = 0$, and $x_2 = 0$, i.e., 0 pounds of the 20-8-8 fertilizer, 0 pounds of the 4-8-4 fertilizer, and 0 pounds of the 4-4-2 fertilizer, which is always the case with the initial tableau: everything is the slack. The business interpretation of this basic solution is this: do not manufacture anything, keep it all hoarded at the warehouse. Whether this makes sense depends on our economic outlook.

Let us translate the above conceptual exercise into Python to make it more concrete. We can use a dictionary to represent the in-variables (i.e., entered variables). We can use an array, too, but the dictionary is more flexible especially in higher-dimensional spaces.

```
in_vars = {0:3, 1:4, 2:5}
>>> in_vars[0]
3
>>> in_vars[1]
4
>>> in_vars[2]
5
```

This representation means that the 0-th in-variable (or, equivalently, the variable in row 0) is x_3, the 1st in-variable is x_4, and the 2-nd in-variable is x_5, i.e., the slacks. The tableau's matrix on which the simplex algorithm will operate can be represented as a numpy array.

```
import numpy as np
m = np.array([[20,  4,  4, 1, 0, 0, 6000],
              [8,   8,  4, 0, 1, 0, 10000],
              [8,   4,  2, 0, 0, 1, 4000],
              [-3, -8, -6, 0, 0, 0, 0]],
             dtype=float)
```

The tableau can now be represented as a Python 2-tuple that consists of the dictionary with the in-variables and the tableau's matrix, on which the pivoting operations will be done. We can, technically, define another tuple that contains the column variables' names. However, since we simplified our tableau notation to use only one variable x with subscripts, the column variable names can be generated from the column indices as and when necessary. So, here is the initial tableau for this problem.

```
>>> tab = (in_vars, m)
>>> tab
({0: 3, 1: 4, 2: 5},
 array([[ 2.e+01,  4.e+00,  4.e+00,  1.e+00,  0.e+00,  0.e+00,  6.e+03],
        [ 8.e+00,  8.e+00,  4.e+00,  0.e+00,  1.e+00,  0.e+00,  1.e+04],
        [ 8.e+00,  4.e+00,  2.e+00,  0.e+00,  0.e+00,  1.e+00,  4.e+03],
        [-3.e+00, -8.e+00, -6.e+00,  0.e+00,  0.e+00,  0.e+00,  0.e+00]]))
>>> tab[0]
{0: 3, 1: 4, 2: 5}
>>> tab[1]
array([[ 2.e+01,  4.e+00,  4.e+00,  1.e+00,  0.e+00,  0.e+00,  6.e+03],
       [ 8.e+00,  8.e+00,  4.e+00,  0.e+00,  1.e+00,  0.e+00,  1.e+04],
       [ 8.e+00,  4.e+00,  2.e+00,  0.e+00,  0.e+00,  1.e+00,  4.e+03],
       [-3.e+00, -8.e+00, -6.e+00,  0.e+00,  0.e+00,  0.e+00,  0.e+00]])
```

The first element of the tableau is a dictionary and the second one is a 2D numpy array. Let us play with this tableau programmatically. You may want to refer to the tables at the top of page 3 as you interact with your Python interpreter.

```
>>> tab[0][0]
3
### the 0th row variable is x3
>>> tab[0][1]
4
### the 1st row variable is x4
>>> tab[0][2]
5
### the 2nd row variable is x5
```

What is the value of the 0th row variable, i.e., x_3, in the basic solution in the tableau `tab`?

```
>>> tab[1][0][6]
6000.0
>>> tab[1][0][6] == 6.e+03
True
```

What is the value of the 1st row variable, i.e., x_4, in the basic solution in the tableau `tab`?

```
>>> tab[1][1][6]
10000.0
>>> tab[1][1][6] == 1.e+4
True
```

What is the value of the 2nd row variable, i.e., x_5, in the basic solution in the tableau `tab`?

```
>>> tab[1][2][6]
4000.0
>>> tab[1][2][6] == 4.e+3
True
```

What is the value of the objective function p in the tableaue `tab`?

```
>>> tab[1][3][6]
0.0
>>> tab[1][3][6] == 0.e+00
True
```

Problem 1 (3 points)

Implement the function `simplex(tab)` that takes a tableau represented as a 2-tuple explained in the Coding Lab section above, runs the simplex algorithm on the tableau, and returns two values: the modified tableau and the Boolean variable that is `True` when the returned tableau contains a solution and `False` when the returned tableau does not contain a solution (i.e., the problem formulated by the initial tableau has no solution).

Below is the output of my implementation of `simplex()` applied to `tab` in the previous section. In my trace, `evc` stands for "entering variable column" and `dvr` stands for "departing variable row." Recall that these two numbers identify the location of the pivot on which the pivoting operation is done. When no entering variable can be found, `evc` is equal to -1. When no departing variable can be found, `dvc` is equal to -1. I display, for debugging purposes, the new tableau (i.e., the dictionary and the matrix) after each pivoting operation. The function `display_solution_from_tab()` is in `cs3430_s24_hw04.py`. The function is explained below.

```
>>> from cs3430_s24_hw04 import simplex, display_solution_from_tab
>>> in_vars = {0:3, 1:4, 2:5}
>>> m = np.array([[20, 4, 4, 1, 0, 0, 6000],
...               [8, 8, 4, 0, 1, 0, 10000],
...               [8, 4, 2, 0, 0, 1, 4000],
...               [-3,-8,-6,0, 0, 0, 0]], dtype=float)
### my implementation of simplex() is destructive, so
### let me make a copy of the tableau.
>>> tab_copy = (in_vars.copy(), m.copy())
>>> final_tab, solved = simplex(tab)
```

```
>>> final_tab, solved = simplex(tab)
in vars: {0: 3, 1: 4, 2: 5}
mat:
[[ 2.e+01  4.e+00  4.e+00  1.e+00  0.e+00  0.e+00  6.e+03]
 [ 8.e+00  8.e+00  4.e+00  0.e+00  1.e+00  0.e+00  1.e+04]
 [ 8.e+00  4.e+00  2.e+00  0.e+00  0.e+00  1.e+00  4.e+03]
 [-3.e+00 -8.e+00 -6.e+00  0.e+00  0.e+00  0.e+00  0.e+00]]
```

```
evc = 1
dvr = 2
pivoting dvr=2, evc=1
in vars: {0: 3, 1: 4, 2: 1}
mat:
[[ 1.2e+01  0.0e+00  2.0e+00  1.0e+00  0.0e+00 -1.0e+00  2.0e+03]
 [-8.0e+00  0.0e+00  0.0e+00  0.0e+00  1.0e+00 -2.0e+00  2.0e+03]
 [ 2.0e+00  1.0e+00  5.0e-01  0.0e+00  0.0e+00  2.5e-01  1.0e+03]
 [ 1.3e+01  0.0e+00 -2.0e+00  0.0e+00  0.0e+00  2.0e+00  8.0e+03]]
evc = 2
dvr = 0
pivoting dvr=0, evc=2
in vars: {0: 2, 1: 4, 2: 1}
mat:
[[ 6.0e+00  0.0e+00  1.0e+00  5.0e-01  0.0e+00 -5.0e-01  1.0e+03]
 [-8.0e+00  0.0e+00  0.0e+00  0.0e+00  1.0e+00 -2.0e+00  2.0e+03]
 [-1.0e+00  1.0e+00  0.0e+00 -2.5e-01  0.0e+00  5.0e-01  5.0e+02]
 [ 2.5e+01  0.0e+00  0.0e+00  1.0e+00  0.0e+00  1.0e+00  1.0e+04]]
evc = -1
>>> final_tab
({0: 2, 1: 4, 2: 1},
  array([[ 6.0e+00,  0.0e+00,  1.0e+00,  5.0e-01,  0.0e+00, -5.0e-01,
          1.0e+03],
        [-8.0e+00,  0.0e+00,  0.0e+00,  0.0e+00,  1.0e+00, -2.0e+00,
          2.0e+03],
        [-1.0e+00,  1.0e+00,  0.0e+00, -2.5e-01,  0.0e+00,  5.0e-01,
          5.0e+02],
        [ 2.5e+01,  0.0e+00,  0.0e+00,  1.0e+00,  0.0e+00,  1.0e+00,
          1.0e+04]]))
>>> solved
True
>>> display_solution_from_tab(final_tab)
x2 = 1000.0
x4 = 2000.0
x1 = 500.0
p = 10000.0
```

Here is a brief explanation of what `display_solution_from_tab()` does. The scientific notation is great, but somewhat hard to work with while debugging (in my humble opinion). Nothing's wrong with it, it just takes some getting used to. So, we can write a couple of functions to display the results better. The first one is getting the solution from a given tableau. The second one displays it. The first function function, `get_solution_from_tab()`, puts the tableau's solution into a dictionary mapping each in-variable to its value in the B.S. column. The value of the objective function is mapped to by the key `'p'`. The second function, `display_solution_from_tab()`, prints the dictionary returned by `get_solution_from_tab()` in a more palatable manner.

```
def get_solution_from_tab(tab):
    in_vars, mat = tab[0], tab[1]
    nr, nc = mat.shape
    sol = {}
    for k, v in in_vars.items():
        sol[v] = mat[k,nc-1]
    sol['p'] = mat[nr-1,nc-1]
```

```
    return sol

def display_solution_from_tab(tab):
    sol = get_solution_from_tab(tab)
    for var, val in sol.items():
        if var == 'p':
            print('p\t=\t{}'.format(val))
        else:
            print('x{}\t=\t{}'.format(var, val))
```

Now we can get a solution from final tab.

```
>>> display_solution_from_tab(final_tab)
x2 = 1000.0
x4 = 2000.0
x1 = 500.0
p = 10000.0
```

This is way better, in my opinion, because we can see right away that the maximum value of the objective function found by the simplex algorithm is 10,000. We also know that the basic solution that corresponds to this value is $x_1 = 500.0$, $x_2 = 1000.0$, and $x_4 = 2000.0$. Recall that our mapping convention is: $x = x_0$, $y = x_1$, $z = x_2$, $u = x_3$, $v = x_4$, and $w = x_5$. Thus, the optimal production schedule for this company is to produce no 20-8-8 fertilizer, 500x100 = 50,000 pounds of the 4-8-4 fertilizer, and 1000x100 = 100,000 pounds of the 4-4-2 fertilizer. Note that in this solution found by the simplex algorithm the slack variable x_4 is equal to 2000, which means that there will be 2,000 pounds of phosphate left over. That's the slack that the company will have to live with.

Let us solve another SMP with our implementation of `simplex()`: maximize $p = 10x + 6y + 2z$ subject to

1. $x \geq 0$;
2. $y \geq 0$;
3. $z \geq 0$;
4. $2x + 2y + 3z \leq 160$;
5. $5x + y + 10z \leq 100$.

We map x to x_0, y to x_1, z to x_2 and introduce two slack variables, x_3 and x_4, for the last two constraints. We set up the initial tableau.

```
>>> from cs3430_s24_hw04 import *
>>>
>>> in_vars = {0:3, 1:4}
>>> m = np.array([[2,    2,   3, 1, 0, 160],
...                   [5,    1,  10, 0, 1, 100],
...                   [-10, -6,  -2, 0, 0, 0]],
...                  dtype=float)
>>> tab = (in_vars, m)
>>> tab_copy = (in_vars.copy(), m.copy())
>>> final_tab, solved = simplex(tab)
in vars: {0: 3, 1: 4}
mat:
```

```
[[  2.   2.   3.   1.   0. 160.]
 [  5.   1.  10.   0.   1. 100.]
 [-10.  -6.  -2.   0.   0.   0.]]
evc = 0
dvr = 1
pivoting dvr=1, evc=0
in vars: {0: 3, 1: 0}
mat:
[[  0.    1.6  -1.    1.   -0.4 120. ]
 [  1.    0.2   2.    0.    0.2  20. ]
 [  0.   -4.   18.    0.    2.  200. ]]
evc = 1
dvr = 0
pivoting dvr=0, evc=1
in vars: {0: 1, 1: 0}
mat:
[[ 0.000e+00  1.000e+00 -6.250e-01  6.250e-01 -2.500e-01  7.500e+01]
 [ 1.000e+00  0.000e+00  2.125e+00 -1.250e-01  2.500e-01  5.000e+00]
 [ 0.000e+00  0.000e+00  1.550e+01  2.500e+00  1.000e+00  5.000e+02]]
evc = -1
>>> solved
True
>>> display_solution_from_tab(final_tab)
x1 = 75.0
x0 = 5.0
p = 500.0
```

Let us test `simplex()` on a tableau that has no solution. This tableau has 3 decision variables (i.e., x_0, x_1, x_2) and two slacks (i.e., x_3 and x_4).

```
>>> in_vars = {0:3, 1:4}
>>> m = np.array([[1,  -1,  1, 1, 0, 5],
                  [2,   0, -1, 0, 1, 10],
                  [-1, -2, -1, 0, 0, 0]],
                 dtype=float)
>>> tab = (in_vars, m)
>>> tab_copy = (in_vars.copy(), m.copy())
>>> final_tab, solved = simplex(tab)
in vars: {0: 3, 1: 4}
mat:
[[ 1. -1.  1.  1.  0.  5.]
 [ 2.  0. -1.  0.  1. 10.]
 [-1. -2. -1.  0.  0.  0.]]
evc = 1
dvr = -1
>>> solved
False
```

Why is `solved False`? Because in the initial tableau the algorithm fails to find the departing variable in the column of the most negative entry (-2) in the p-row. Thus, the problem has no solution.

Remember the Ted's Toys problem? If not, you can take a look at Slide 6 in Lecture 05. Let us solve it with `simplex()`.

```
>>> in_vars = {0:2, 1:3}
>>> m = np.array([[ 4,  3,  1, 0, 480],
                  [ 3,  6,  0, 1, 720],
                  [-5, -4,  0, 0, 0]],
                 dtype=float)
>>> tab = (in_vars, m)
### I skip making a copy.
>>> tab, solved = simplex(tab)

in vars: {0: 2, 1: 3}
mat:
[[  4.   3.   1.   0. 480.]
 [  3.   6.   0.   1. 720.]
 [ -5.  -4.   0.   0.   0.]]
evc = 0
dvr = 0
pivoting dvr=0, evc=0

in vars: {0: 0, 1: 3}
mat:
[[ 1.00e+00  7.50e-01  2.50e-01  0.00e+00  1.20e+02]
 [ 0.00e+00  3.75e+00 -7.50e-01  1.00e+00  3.60e+02]
 [ 0.00e+00 -2.50e-01  1.25e+00  0.00e+00  6.00e+02]]
evc = 1
dvr = 1
pivoting dvr=1, evc=1

in vars: {0: 0, 1: 1}
mat:
[[ 1.00000000e+00  0.00000000e+00  4.00000000e-01 -2.00000000e-01
   4.80000000e+01]
 [ 0.00000000e+00  1.00000000e+00 -2.00000000e-01  2.66666667e-01
   9.60000000e+01]
 [ 0.00000000e+00  0.00000000e+00  1.20000000e+00  6.66666667e-02
   6.24000000e+02]]
evc = -1

in_vars = {0: 0, 1: 1}
tab = [[ 1.00000000e+00  0.00000000e+00  4.00000000e-01 -2.00000000e-01
   4.80000000e+01]
 [ 0.00000000e+00  1.00000000e+00 -2.00000000e-01  2.66666667e-01
   9.60000000e+01]
 [ 0.00000000e+00  0.00000000e+00  1.20000000e+00  6.66666667e-02
   6.24000000e+02]]

>>> solved
True
>>> display_solution_from_tab(tab)
x0 = 48.0
x1 = 96.0
p = 624.0
```

Problem 2: (2 points)

Use your implementation of `simplex()` to solve the following problems and save your solutions in the appropriate stubs in `cs3430_s24_hw04.py`.

Problem 2.1

Maximize $p = 2x + 3y$ subject to

1. $x \geq 0$;
2. $y \geq 0$;
3. $3x + 8y \leq 24$;
4. $6x + 4y \leq 30$.

Problem 2.2

Maximize $p = x$ subject to

1. $x \geq 0$;
2. $y \geq 0$;
3. $x - y \leq 4$;
4. $-x + 3y \leq 4$.

Problem 2.3

The Brown Brothers Box Company is bidding on a new contract to manufacture boxes for computers, printers, and paint. A computer box requires 12 square feet of heavy-duty cardboard, 18 square feet of regular cardboard, and 15 square feet of white facing paper. A printer box requires 6 square feet of heavy-duty cardboard, 12 square feet of regular cardboard, and 8 square feet of white facing paper. A paint box requires 10 square feet of regular cardboard. Current arrangements with the suppliers allow the company the following weekly allocations: 1,500 square feet of heavy-duty cardboard, 2,500 square feet of regular cardboard, and 2,000 square feet of facing paper. The profit is \$1.50 for a computer box, \$0.80 for a printer box, and \$0.25 for a paint box. Solve for the weekly allocation of resources which yields maximum profit.

The unit test for this problem is written with the following assumption about the decision varaibles: x_0 – the number of computer boxes; x_1 - the number of printer boxes; and x_2 is the number of paint boxes.

What to Submit

Save your coding solutions in `cs3430_s24_hw04.py` included in the zip and submit it in Canvas. My unit tests are in `cs3430_s21_hw04_uts.py`. My output for each unit test is given in a multi-line comment after the test.

Happy Hacking!