INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

PROGRAMACIÓN LINEAL

Proyecto final

Formulación y solución de problemas de programación lineal

Integrantes del equipo	Clave única
Karen Arteaga Mendoza	190161
Federico Santacruz González	190438
Leopoldo Rodríguez Díaz Infante	189584
José Alberto Márquez Luján	187917
Santiago Fernández del Castillo Sodi	189210

Profesor:

Dr. Edgar Possani Espinosa

26 de noviembre de 2022

1. Programación del Método Simplex

1.1. Pseudocódigo y explicación

La programación del Método Simplex se llevó a cabo en el lenguaje de programación Python. Se divide en cuatro funciones importantes:

1. SOLVER es la función principal: aquí se manda a llamar al resto de los métodos. Recibe como entrada los datos del problema en su formulación estándar y regresa la tabla final y el valor de la función objetivo evaulada en la solución óptima.

2. MATRIZT0 es una función sencilla que se utiliza para construir la tabla cero del problema a partir de los datos y con base en el Método de la Gran M.

3. REGLADEBLAND es una función que recibe como parámetro una tabla Simplex y, basándose en la Regla de Bland, decide si la tabla es final, el problema no es acotado o hay que pivotear; si es final, regresa $(-1, -1)$; si el problema no está acotado, regresa $(-1, -2)$, y, si hay que realizar un paso más, regresa la posición (fila y columna) del elemento sobre el cual se va a pivotear en el siguiente paso. Esta función se manda a llamar solamente desde PIVOTEO.

4. PIVOTEO Dada una tabla Simplex, esta función pivotea sobre el elemento que dictamina la regla de Bland y regresa el resultado. Si no se pudo pivotear, ya sea porque es tabla final o porque el problema no está acotado, regresa como valor el que obtiene de la función REGLADEBLAND. Si pudo pivotear, regresa un número 1.

A continuación se presenta el pseudocódigo de la función SOLVER. La función primero obtiene la tabla cero del problema al mandar llamar a MATRIZT0. Posteriormente obtiene la tabla Simplex inicial al asegurarse de que la base se encuentra en la tabla. Después, intenta pivotear sobre la tabla hasta que encuentre que el problema no es acotado o bien encuentre la solución del problema. Por último, regresa la última tabla a la que llegó y el valor de la función objetivo evaluada en la solución o NAN si el problema no está acotado.

Algorithm 1 Función SOLVER

```
function SOLVER(A, b, c, M) returns t_1, z
    t_0 ← MATRIZT0(A, b, c, M)
    for i in RANGE(LEN(A)) do
        t_0[−1] ← t_0[−1] + t_0[i] ∗ (−M)
    end for
    t_1, z ← PIVOTEO(t_0)
    while z ≥ 0 do
        t_1, z ← PIVOTEO(t_1)
    end while
    if no acotado then
        return t_1, NaN
    end if
    return t_1, (−1) ∗ t_1[−1][−1]
end function
```

Para la función MATRIZT0 solamente se manipulan los parámetros con la librería `numpy`; al tratarse de pocas líneas y de procedimiento sencillo, no incluiremos un pseudocódigo. Basta mencionar que pegamos los costos relativos hasta abajo de la tabla y, a la derecha, agregamos unas columnas que corresponden a las variables añadidas por el Método de la Gran M. La última columna corresponde al vector b y se inicializa el valor de la función objetivo en cero.

La función REGLADEBLAND es un poco más interesante. Como se mencionó al principio, recibe como parámetro una tabla Simplex. Primero busca los índices de los costos relativos en los que los costos son estrictamente negativos. Si no encuentra ninguno, entonces regresa $(-1, -1)$ pues la tabla es final. Si encuentra alguno, entonces almacena como columna de salida el índice que esté más a la izquierda. Posteriormente, recorre b_i y y_i, donde b_i es el elemento de hasta la derecha de la tabla en la i-ésima fila y y_i es el elemento correspondiente, pero de la columna de salida. Si y_i es estrictamente postivo, entonces almacena en un arreglo el valor de b_i/y_i; si no, almacena un -1. Luego busca en ese arreglo si hay valores mayores o iguales a cero; si hay, regresa el índice que corresponde a la fila de ese elemento; si no hay, entonces regresa $(-1, -2)$, pues estaríamos tratando con un problema no acotado. El pseudocódigo se presenta a continuación:

Algorithm 2 Función REGLADEBLAND

```
function REGLADEBLAND(tablaSimplex) returns renglonSal, colSal
    cr ← costos relativos tablaSimplex
    busq ← índices donde cr < 0
    if LEN(busq) == 0 then                    ▷ todos los costos relativos son positivos
        return (−1, −1)
    else
        colSal ← busq[0]
    end if
    b_k ← tablaSimplex[:, -1]
    y_k ← tablaSimplex[:, colSal]
    by ← []
    for b_i, y_i in (b_k, y_k) do
        if y_i > 0 then
            by.APPEND(b_i/y_i)
        else
            by.APPEND(−1)
        end if
    end for
    valid ← índices donde by ≥ 0
    if LEN(valid) == 0 then                   ▷ el problema no está acotado
        return (−1, −2)
    end if
    renglonSal ← argmin{by | by> 0}
    return renglonSal, colSal
end function
```

Por último, se presenta el pseudocódigo de la función PIVOTEO, la cual recibe una tabla Simplex e intenta aplicar la Regla de Bland con la función correspondiente: si no puede pivotear, regresa la tabla Simplex junto con el número negativo que indica la razón por la que se terminó el método; si puede pivotear, divide toda la fila escogida entre el valor que sobre el cual se va a pivotear (para que éste sea 1) y luego, para cada renglón que no es el escogido, resta el valor correspondiente multiplicado por el valor del pivoteo.

Algorithm 3 Función PIVOTEO

```
function PIVOTEO(tablaSimplex) returns tablaSimplex, num
    renglonSal, colSal ← REGLADEBLAND(tablaSimplex)
    if colSal < 0 then                                ▷ hay una razón para detenerse
        return tablaSimplex, colSal
    end if
    m ← LEN(tablaSimplex)
    valorPivoteo ← tablaSimplex[renglonSal][colSal]
    tablaSimplex[renglonSal] ← tablaSimplex[renglonSal] / valorPivoteo
    for i in RANGE(m) do
        if i ≠ renglonSal and tablaSimplex[i][colSal] ≠ 0 then
            tablaSimplex -= tablaSimplex[i][colSal] * tablaSimplex[renglonSal]
        end if
    end for
    return tablaSimplex, 1
end function
```

1.2. Resultados del código

1.2.1. Pruebas diseñadas por el equipo

[6]:
```python
c = np.array([1, 1, 0, 0])
A = np.array([[3, 4, 1, 0],
            [ 2,-1, 0,-1]])
b = np.array([20,2])
M = 100

t1, z = solver(A, b, c, M)
print(t1)
print(f"El valor de la función objetivo es: {z}")
```

```
[[0.0 5.5 1.0 1.5 1.0 -1.5 17.0]
[1.0 -0.5 0.0 -0.5 0.0 0.5 1.0]
[0.0 1.5 0.0 0.5 100.0 99.5 -1.0]]

El valor de la función objetivo es: 1.0000000000002132
```

[7]:
```python
c1 = np.array([0, -9, -1, 0, 2, 1])
A1 = np.array([[0, 5, 50, 1, 1, 0],
            [1, -15, 2, 0, 0, 0],
            [0, 1, 1, 0, 1, 1]])
b1 = np.array([10, 2, 6])
t1, z1 = solver(A1, b1, c1, M)
print(t1)
print(f"\nEl valor de la función objetivo es: {z1}")
```

```
[[0.0 1.0 10.0 0.2 0.2 0.0 0.2 0.0 0.0 2.0]
[1.0 0.0 152.0 3.0 3.0 0.0 3.0 1.0 0.0 32.0]
```

```
[0.0 0.0 -9.0 -0.2 0.8 1.0 -0.2 0.0 1.0 4.0]
[0.0 0.0 98.0 2.0 3.0 0.0 102.0 100.0 99.0 14.0]]

El valor de la función objetivo es: -14.0
```

[8]:
```python
c2 = np.array([-3, 1, 0, 0])
A2 = np.array([[-1, 1, 1, 0],
               [2, 2, 0, -1]])
b2 = np.array([5, 4])
t2, z2 = solver(A2, b2, c2, M)
print(t2)
print(f"\nEl valor de la función objetivo es: {z2}")
```

```
PROBLEMA NO ACOTADO; última versión de la tabla:
[[0.0 2.0 1.0 -0.5 1.0 0.5 7.0]
[1.0 1.0 0.0 -0.5 0.0 0.5 2.0]
[0.0 4.0 0.0 -1.5 100.0 101.5 6.0]]

El valor de la función objetivo es: nan
```

1.2.2. Pruebas diseñadas por el profesor

TODO

1.3. Posibles áreas de mejora

Creemos que el código debería ser un poco más versátil a la hora de aceptar problemas como entrada. Por ahora, solamente acepta problemas de programación lineal en su forma estándar. Quizás en un futuro no tan lejano podría aceptar también restricciones de desigualdades y que automáticamente agruegue las variables de holgura necesarias. También nos gustaría incluir un sistema que acepte variables libres y valores absolutos en las variables; esta parte del código debería ser similar. Una mejoría también podría ser que vaya imprimiendo los pasos que está realizando, de manera que un estudiante sea capaz de ingresar un problema y seguir lo que el programa hizo para poder detectar algún error.

2. Anexo: el código

[1]:
```python
import numpy as np
np.set_printoptions(formatter={'float': lambda x: "{0:0.3f}".format(x)})␣
↪# Para que enseñe nada más 3 decimales
```

[2]:
```python
def matriz_t0(A, b, c, M):
    '''
    Método para obtener la tabla cero del problema usando el método
    de la Gran M.

    EJEMPLO DE USO:
```

```
    >>> A = np.array([[3, 4, 1, 0],
                      [ 2,-1, 0,-1]])
    >>> b = np.array([20,2])
    >>> c = np.array([1, 1, 0, 0])
    >>> M = 100
    >>> matriz_t0(A, b, c, M)
    array([[  3.,   4.,   1.,   0.,   1.,   0.,  20.],
           [  2.,  -1.,   0.,  -1.,   0.,   1.,   2.],
           [  1.,   1.,   0.,   0., 100., 100.,   0.]])
    '''
    m = len(A)
    canon = np.eye(m, dtype=int)                        # Matriz identidad␣
↪mxm
    mat1 = np.concatenate((A,canon), axis=1)            # Pegamos A con la␣
↪identidad
    cr = np.append(c,[M]*m)                             # Vector de costos␣
↪relativos
    mat1 = np.concatenate((mat1, np.array([cr])))       # Pegamos la matriz␣
↪con los costos relativos
    b_ext = np.array([np.append(b, 0)])                 # Construcción del␣
↪vector b

    # Regresamos la matriz extendida con b
    return np.concatenate((mat1, b_ext.T), axis=1).astype('float64')
```

[3]:
```
def reglaDeBland(tablaSimplex):
    '''
    Queremos la columna más a la izquierda con cr < 0.
    Si hay empate en el criterio de la variable de salida,
    elegimos la más arriba
    '''
    cr = tablaSimplex[-1][:-1] # costos relativos
    busq = np.where(cr < 0)[0]

    if len(busq) == 0: # No encontró; fin del problema
        return -1, -1
    else:
        colSal = busq[0]

    bk = tablaSimplex[:, -1]
    yk = tablaSimplex[:, colSal]

    by = np.empty(0)
    for b,y in zip(bk,yk):
        if y > 0:
            by = np.append(by, b/y)
        else:
            by = np.append(by, -1)
```

```
    valid = np.where(by >= 0)[0]

    if len(valid) == 0: # Todas las variables son menores que cero
        return -1, -2

    renglonSal = valid[by[valid].argmin()]

    return renglonSal, colSal
```

[4]:
```
def pivoteo(tablaSimplex):
    '''
    Dada una tabla Simplex, este método pivotea sobre el elemento
    que dictamina la regla de Bland y regresa el resultado.
    '''
    renglonSal, colSal = reglaDeBland(tablaSimplex)

    if colSal < 0: # Condiciones para detenerse
        return tablaSimplex, colSal

    m = len(tablaSimplex)

    valorPivoteo = tablaSimplex[renglonSal][colSal]
    tablaSimplex[renglonSal] = tablaSimplex[renglonSal] / valorPivoteo

    for i in range(m):
        if i != renglonSal and tablaSimplex[i][colSal] != 0:
            tablaSimplex[i] -= tablaSimplex[i][colSal] *␣
 ↪tablaSimplex[renglonSal]

    return tablaSimplex, 1
```

[5]:
```
def solver(A,b,c,M=100):
    '''
    Método para resolver un PPL planteado en su forma estándar. Utiliza
    el método de la Gran M y Simplex. Regresa la tabla en su forma final
    y el resultado de la función objetivo.
    '''
    t0 =  matriz_t0(A, b, c,M)

    for i in range(len(A)):
        t0[-1] = t0[i]*(-M) + t0[-1]

    print(t0)
    print('')

    t1, z = pivoteo(t0) # Aquí z es la columna de salida y la usamos␣
 ↪como control para saber si terminó.

    while z >= 0:
        t1, z = pivoteo(t1)
```

```
    if z == -2: # no está acotado
        print("PROBLEMA NO ACOTADO; última versión de la tabla:")
        return t1, np.nan

    return t1, (-1)*t1[-1][-1]
```