Open in Colab

Math 210

Aaron Graybill

Midterm 2

4/30/21

Imports

In [50]:
```
import numpy as np
from scipy.optimize import linprog
import pandas as pd
import itertools
```

In [51]:
```
def print_tableau(a,indep_names,dep_names,indep_names_dual,dep_names_dual):
#
# Given matrix "a" and lists of variables names "indep_names" and "dep_names",
#   and (for the dual) "indep_names_dual" and "dep_names_dual",
# this function prints the matrix and labels in standard tableau format
# (including adding the -1, the minus signs in the last column, and labeling the lower-right as obj)
#
# First, check the inputs: indep_names and dep_names_dual should be one shorter than the number of columns of A
#                          dep_names and indep_names_dual should be one shorter than the number of rows of A
#
    nrows = a.shape[0]    # use the shape function to determine number of rows and cols in A
    ncols = a.shape[1]
    nindep = len(indep_names)
    nindep_dual = len(indep_names_dual)
    ndep = len(dep_names)
    ndep_dual = len(dep_names_dual)
    if nindep != ncols-1:
        print("WARNING: # of indep vbles should be one fewer than # columns of matrix")
    if ndep != nrows-1:
        print("WARNING: # of dep vbles should be one fewer than # rows of matrix")
    if nindep_dual != nrows-1:
        print("WARNING: # of indep dual vbles should be one fewer than # rows of matrix")
    if ndep_dual != ncols-1:
        print("WARNING: # of dep dual vbles should be one fewer than # columns of matrix")
# Now do the printing (uses a variety of formatting techniques in Python)
    print("          ",end="")       # On first line, leave blank space so we can fit in dual labels lower down
    for j in range(ncols-1):                    # Print the independent variables in the first row
        print(indep_names[j].rjust(10),end="")  # rjust(10) makes fields 10 wide and right-justifies;
                                                #    the end command prevents newline)
    print("        -1")                         # Tack on the -1 at the end of the first row
    for i in range(nrows-1):
        print(indep_names_dual[i].rjust(10),end="")
        for j in range(ncols):                  # Print all but the last row of the matrix
                print("%10.3f" % a[i][j],end="") # The syntax prints in a field 10 wide, showing 3 decimal point
        lab = "= -" + dep_names[i]
        print(lab.rjust(10))
    print("        -1",end="")
    for j in range(ncols):
        print("%10.3f" % a[nrows-1][j],end="")  # Print the last row of the matrix, with label "obj" at end
    lab = "= obj"
    print(lab.rjust(10))
    print("          ",end="")
    for j in range(ncols-1):
        lab = "=" + dep_names_dual[j]
        print(lab.rjust(10),end="")
    print("  =dualobj")
    print(" ")    # Put blank line at bottom
```

In [52]:
```
def pivot(a,pivrow,pivcol,indep_names,dep_names,indep_names_dual,dep_names_dual) :
#
# Given matrix "a", a row number "pivrow" and column number "pivcol",
#  and lists of variable names "indep_names" and "dep_names", this
#  function does three things:
#    (1) outputs the new version of the matrix after a pivot,
#    (2) updates the lists of variable names post-pivot
#    (3) prints the new matrix, including labels showing the variable names
#
```

```
# First, check the inputs: indep_names should be one shorter than the number of columns of A
#                          dep_names should be one shorter than the number of rows of A
#                          you should not be pivoting on the last row or last column
#
    nrows = a.shape[0]    # use the shape function to determine number of rows and cols in A
    ncols = a.shape[1]
    nindep = len(indep_names)
    nindep_dual = len(indep_names_dual)
    ndep = len(dep_names)
    ndep_dual = len(dep_names_dual)
    if nindep != ncols-1:
        print("WARNING: # of indep vbles should be one fewer than # columns of matrix")
    if ndep != nrows-1:
        print("WARNING: # of dep vbles should be one fewer than # rows of matrix")
    if nindep_dual != nrows-1:
        print("WARNING: # of indep dual vbles should be one fewer than # rows of matrix")
    if ndep_dual != ncols-1:
        print("WARNING: # of dep dual vbles should be one fewer than # columns of matrix")
    if pivrow > nrows-1 or pivcol > ncols-1:
        print("WARNING: should not pivot on last row or column")
    newa = a.copy()       # make a copy of A, to be filled in below with result of pivot
    p = a[pivrow-1][pivcol-1]   # identify pivot element
    newa[pivrow-1][pivcol-1] = 1/p   # set new value of pivot element
    # Set entries in p's row
    for j in range(ncols):
        if j != pivcol-1:
            newa[pivrow-1][j]=a[pivrow-1][j]/p;
    # Set entries in p's column
    for i in range(nrows):
        if i != pivrow-1:
            newa[i][pivcol-1]=-a[i][pivcol-1]/p;
    # Set all other entries
    for i in range(nrows):
        for j in range(ncols):
            if i != pivrow-1 and j != pivcol-1:
                r = a[i][pivcol-1]
                q = a[pivrow-1][j]
                s = a[i][j]
                newa[i][j]=(p*s-q*r)/p
    # Now transfer the new tableau into a
    for i in range(nrows) :
        for j in range(ncols) :
            a[i][j] = newa[i][j]
    # Now swap the variable names
    temp = indep_names[pivcol-1]
    indep_names[pivcol-1]=dep_names[pivrow-1]
    dep_names[pivrow-1]=temp
    temp = indep_names_dual[pivrow-1]
    indep_names_dual[pivrow-1]=dep_names_dual[pivcol-1]
    dep_names_dual[pivcol-1]=temp
    print_tableau(newa,indep_names,dep_names,indep_names_dual,dep_names_dual) # Print the matrix with updated la
    return 0;
```

In [53]:
```
def column_delete(a,col_to_remove,indep_names,dep_names,indep_names_dual,dep_names_dual) :
    import numpy as np
    anew = np.delete(a,col_to_remove-1,axis=1)
    del indep_names[col_to_remove-1]
    del dep_names_dual[col_to_remove-1]
    print_tableau(anew,indep_names,dep_names,indep_names_dual,dep_names_dual)
    return anew
```

In [54]:
```
def row_delete(a,row_to_remove,indep_names,dep_names,indep_names_dual,dep_names_dual) :
    import numpy as np
    anew = np.delete(a,row_to_remove-1,axis=0)
    del dep_names[row_to_remove-1]
    del indep_names_dual[row_to_remove-1]
    print_tableau(anew,indep_names,dep_names,indep_names_dual,dep_names_dual)
    return anew
```

In [55]:
```
def target(a) :
    nrows = a.shape[0]    # use the shape function to determine number of rows and cols in "a"
    ncols = a.shape[1]
    import numpy as np
    v = np.empty(ncols-1)
    for i in range(ncols-1):
        v[i]=a[nrows-1,i]
    biggest_c = np.max(v)
    where_is_biggest_c = np.argmax(v)+1
    if biggest_c > 0 :
        return where_is_biggest_c
```

```
    else :
        return -1
```

In [56]:
```
def select(a,pivcolnum) :
    nrows = a.shape[0]    # use the shape function to determine number of rows and cols in A
    ncols = a.shape[1]
# First task: work down the column and record the b/a ratios in a vector v
#     except record -1 if a is negative or zero
    import numpy as np
    v = np.zeros(nrows-1)
    for i in range(nrows-1):
        if a[i,pivcolnum-1]>0 :
            v[i] = a[i,ncols-1]/a[i,pivcolnum-1]
        else :
            v[i] = -1
# Second task: if max b/a > -1, find min b/a by hand (ignoring zero entries in v)
    if np.max(v) > -1 :
        min_so_far = np.max(v)+1  # Initialize min to be for-sure bigger than the min
        for i in range(nrows-1):
            if v[i] > -1 and v[i] < min_so_far :
                min_so_far = v[i]
                where_is_min = i+1   # Add 1 to use human numbering
        return where_is_min      # Once we've scanned v for min, we can return result

    else :          # Otherwise, we find the m
        return -1
```

In [57]:
```
def simplexbf(a,indep_names,dep_names,dual_indep_names,dual_dep_names):
# Run the simplexbf algorithm
# Inputs: np.array "a" (assumed to be basic feasible)
#         lists of variable names indep_names and dep_names (pivot will catch if they're wrong size)
# Output: -1 if we stop because problem is unbounded, 0 if we continue to a solution
#         -9 if we take too many steps
    nrows = a.shape[0]    # use shape to find # of rows and cols in A
    ncols = a.shape[1]
    print("Starting SimplexBF (will do nothing if solution can already be determined)")
    pivcol = target(a)
    nsteps = 0
    while pivcol > -1 and nsteps < 50:  # Repeat until either solution found or 50 pivots completed
        pivrow = select(a,pivcol)
        if pivrow == -1 :
            return -1    # If select reports -1, problem is unbounded, so exit this function
        else :
            pivot(a,pivrow,pivcol,indep_names,dep_names,dual_indep_names,dual_dep_names)
            nsteps=nsteps+1
            pivcol = target(a)
    if nsteps >= 50:
        return -9    # we took too many pivots
    else:
        return 0
```

In [58]:
```
def targetnbf(a):
    nrows = a.shape[0]
    ncols = a.shape[1]
    import numpy as np
    checkrow = nrows-2
    while a[checkrow,ncols-1] >= -0.00000001 :
        if checkrow == 0 :   # if still in the "while" and at the top,
            return -1        # all the b's were >= 0, so return -1
        else :
            checkrow = checkrow-1
    return checkrow+1    # if we exit the "while", we found a negative
#                            b, so return current row # (in human numbering)
```

In [59]:
```
def selectnbf(a,targetrow) :
#  Given inputs "a" (tableau as an np.array, numbers only, no labels)
#       and "targetrow" (a row that has a negative b; start-at-1 numbering assumed),
#  computes a pivot that could be chosen by SimplexNBF and
#  outputs "pivrow" and "pivcol", the row and column (start-at-1 numbering) of that pivot
#  If the targeted row has no negative aij, returns -2 for both pivrow and pivcol
    nrows = a.shape[0]
    ncols = a.shape[1]
    import numpy as np
    targetrow = targetrow-1   # convert to start-at-0
    pivcol = ncols-2     # column index of last aij
    while a[targetrow,pivcol] >= 0 :
        if pivcol == 0 :   # if pivcol makes it to zero, all aij
            return [-2,-2]   # in this row were >= 0, so problem infeasible
        else :
            pivcol = pivcol-1
```

```
    minsofar = a[targetrow,ncols-1]/a[targetrow,pivcol] # we found a negative aij
    pivrow = targetrow
    for i in range(targetrow+1,nrows-2):   # now check below it for a smaller bi/aij with aij>0
        if a[i,pivcol]>0 and a[i,ncols-1]/a[i,pivcol] < minsofar :
            minsofar = a[i,ncols-1]/a[i,pivcol]
            pivrow = i
    return [pivrow+1,pivcol+1]   # Return result (shifted to start-at-1 numbering)
```

In [60]:
```
def simplexnbf(a,indep_names,dep_names,dual_indep_names,dual_dep_names):
# Run the simplexnbf algorithm
# Inputs: np.array "a"
#         lists of variable names indep_names and dep_names (pivot will catch if they're wrong size)
# Output: -2 if we stop because problem is infeasible, 0 if we stop at a basic feasible tableau
#.        -9 if we take too many pivots
#    (Also, the tableau "a" and variable-lists are updated with each pivot)
    nrows = a.shape[0]
    ncols = a.shape[1]
    print("Starting SimplexNBF (will do nothing if already basic feasible)")
    nsteps = 0
    targetrow = targetnbf(a)
    while targetrow > -1 and nsteps < 50:  # Repeat until either basic feasible tableau produced or 50 pivots co
        [pivrow,pivcol] = selectnbf(a,targetrow)
        if pivrow == -2 :
            return -2    # If selectnbf reports -2, problem is infeasible, so exit this function
        else :
            pivot(a,pivrow,pivcol,indep_names,dep_names,dual_indep_names,dual_dep_names)
            nsteps=nsteps+1
            targetrow = targetnbf(a)
    if nsteps >= 50:
        return -9    # took too many pivots
    else:
        return 0
```

In [61]:
```
def simplex(a,indep_names,dep_names,dual_indep_names,dual_dep_names) :
# Runs the simplex algorithm (doing NBF if needed, then BF)
# Inputs: np.array "a"
#         lists of variable names indep_names and dep_names (pivot will catch if they're wrong size)
# Output: -2 if problem is infeasible
#         -1 if problem is unbounded
#         0 if problem has a solution
#    (Also, the tableau "a" and variable-lists are updated with each pivot)
    nrows = a.shape[0]
    ncols = a.shape[1]
    print("Initial tableau")
    print_tableau(a,indep_names,dep_names,dual_indep_names,dual_dep_names)
    code = simplexnbf(a,indep_names,dep_names,dual_indep_names,dual_dep_names)
    if code == -2 :
        print("Problem is infeasible")
        return -2
    elif code == -9 :
        print("SimplexNBF took too many pivots")
    else :
        code = simplexbf(a,indep_names,dep_names,dual_indep_names,dual_dep_names)
        if code == -1 :
            print("Problem is unbounded")
            return -1
        elif code == -9 :
            print("SimplexBF took too many pivots")
        else :
            print("Problem has solution, final tableau is shown above")
            return 0
```

In [62]:
```
def simplexeq(a,k,indep_names,dep_names,dual_indep_names,dual_dep_names) :
# Specialized function to do the "pre-simplex" step to handle tableaus where the
#   first k rows correspond to equality constraints.
    nrows = a.shape[0]
    ncols = a.shape[1]
    print_tableau(a,indep_names,dep_names,dual_indep_names,dual_dep_names)
    for i in range(k) :
        j=0
        pivcol=-1
        for j in range(ncols-1) :
            if abs(a[i,j]) > 0.000001 :
                pivrow=i+1
                pivcol=j+1
                break
        if pivcol == -1 :
            return -3
        else :
            pivot(a,pivrow,pivcol,indep_names,dep_names,dual_indep_names,dual_dep_names)
```

```
            a=column_delete(a,pivcol,indep_names,dep_names,dual_indep_names,dual_dep_names)
            ncols=ncols-1
    code = simplex(a,indep_names,dep_names,dual_indep_names,dual_dep_names)
    return code
```

Problem 1.

a.

This problem asks for a standard implementation of simplex with equality constraints.

The first two constraints are as given and the only other constraints are non-negativity.

We maximize the given expression resulting in the following tableau:

In [62]:

In [63]:
```
a=np.array([[1,1,1,1,1,1,22],
            [1,-2,3,-4,5,-6,0],
            [2,3,5,7,11,13,0]])
a = a.astype(float)
indep_names=["x1","x2","x3","x4","x5","x6"]
dep_names=["0","0"]
indep_names_dual=["y1_circ","y2_circ"]
dep_names_dual=["s1","s2","s3","s4","s5","s6"]
print_tableau(a,indep_names,dep_names,indep_names_dual,dep_names_dual)
```

```
                x1        x2        x3        x4        x5        x6        -1
  y1_circ    1.000     1.000     1.000     1.000     1.000     1.000    22.000      = -0
  y2_circ    1.000    -2.000     3.000    -4.000     5.000    -6.000     0.000      = -0
       -1    2.000     3.000     5.000     7.000    11.000    13.000     0.000     = obj
               =s1       =s2       =s3       =s4       =s5       =s6  =dualobj
```

In [64]:
```
simplexeq(a,2,indep_names,dep_names,indep_names_dual,dep_names_dual)
```

```
                x1        x2        x3        x4        x5        x6        -1
  y1_circ    1.000     1.000     1.000     1.000     1.000     1.000    22.000      = -0
  y2_circ    1.000    -2.000     3.000    -4.000     5.000    -6.000     0.000      = -0
       -1    2.000     3.000     5.000     7.000    11.000    13.000     0.000     = obj
               =s1       =s2       =s3       =s4       =s5       =s6  =dualobj

                0        x2        x3        x4        x5        x6        -1
       s1    1.000     1.000     1.000     1.000     1.000     1.000    22.000     = -x1
  y2_circ   -1.000    -3.000     2.000    -5.000     4.000    -7.000   -22.000      = -0
       -1   -2.000     1.000     3.000     5.000     9.000    11.000   -44.000     = obj
          =y1_circ       =s2       =s3       =s4       =s5       =s6  =dualobj

               x2        x3        x4        x5        x6        -1
       s1    1.000     1.000     1.000     1.000     1.000    22.000     = -x1
  y2_circ   -3.000     2.000    -5.000     4.000    -7.000   -22.000      = -0
       -1    1.000     3.000     5.000     9.000    11.000   -44.000     = obj
               =s2       =s3       =s4       =s5       =s6  =dualobj

                0        x3        x4        x5        x6        -1
       s1    0.333     1.667    -0.667     2.333    -1.333    14.667     = -x1
       s2   -0.333    -0.667     1.667    -1.333     2.333     7.333     = -x2
       -1    0.333     3.667     3.333    10.333     8.667   -51.333     = obj
          =y2_circ       =s3       =s4       =s5       =s6  =dualobj

               x3        x4        x5        x6        -1
       s1    1.667    -0.667     2.333    -1.333    14.667     = -x1
       s2   -0.667     1.667    -1.333     2.333     7.333     = -x2
       -1    3.667     3.333    10.333     8.667   -51.333     = obj
               =s3       =s4       =s5       =s6  =dualobj

Initial tableau
               x3        x4        x5        x6        -1
       s1    1.667    -0.667     2.333    -1.333    14.667     = -x1
       s2   -0.667     1.667    -1.333     2.333     7.333     = -x2
       -1    3.667     3.333    10.333     8.667   -51.333     = obj
               =s3       =s4       =s5       =s6  =dualobj

Starting SimplexNBF (will do nothing if already basic feasible)
Starting SimplexBF (will do nothing if solution can already be determined)
               x3        x4        x1        x6        -1
       s5    0.714    -0.286     0.429    -0.571     6.286     = -x5
       s2    0.286     1.286     0.571     1.571    15.714     = -x2
       -1   -3.714     6.286    -4.429    14.571  -116.286     = obj
               =s3       =s4       =s1       =s6  =dualobj
```

```
          x3        x4        x1        x2        -1
s5     0.818     0.182     0.636     0.364    12.000     = -x5
s6     0.182     0.818     0.364     0.636    10.000     = -x6
-1    -6.364    -5.636    -9.727    -9.273  -262.000     = obj
         =s3       =s4       =s1       =s2  =dualobj

Problem has solution, final tableau is shown above
```

Out[64]: 0

The output above indicates that $x_5 = 12$ and $x_6 = 10$ with all others equal to 0. The maximum value of the objective function would then be 262.

b.

I reprint the dual tableau as a reference before describing the dual problem. We have:

In [65]:
```
a=np.array([[1,1,1,1,1,1,22],
            [1,-2,3,-4,5,-6,0],
            [2,3,5,7,11,13,0]])
indep_names=["x1","x2","x3","x4","x5","x6"]
dep_names=["0","0"]
indep_names_dual=["y1_circ","y2_circ"]
dep_names_dual=["s1","s2","s3","s4","s5","s6"]
print_tableau(a,indep_names,dep_names,indep_names_dual,dep_names_dual)
```

```
            x1        x2        x3        x4        x5        x6        -1
y1_circ  1.000     1.000     1.000     1.000     1.000     1.000    22.000      = -0
y2_circ  1.000    -2.000     3.000    -4.000     5.000    -6.000     0.000      = -0
      -1 2.000     3.000     5.000     7.000    11.000    13.000     0.000     = obj
           =s1       =s2       =s3       =s4       =s5       =s6  =dualobj
```

Meaning we have, minimize: $22y_1 + 0y_2$ subject to:

$$
\begin{aligned}
y_1 + y_2 &\geq 2 \\
y_1 - 2y_2 &\geq 3 \\
y_1 + 3y_2 &\geq 5 \\
y_1 - 4y_2 &\geq 7 \\
y_1 + 5y_2 &\geq 11 \\
y_1 - 6y_2 &\geq 13
\end{aligned}
$$

Where both y_1 and y_2 are unconstrained.

The optimal value would be the same as above, 262. We know from the solution above that $s_5 = s_6 = 0$. Meaning we can intersect:

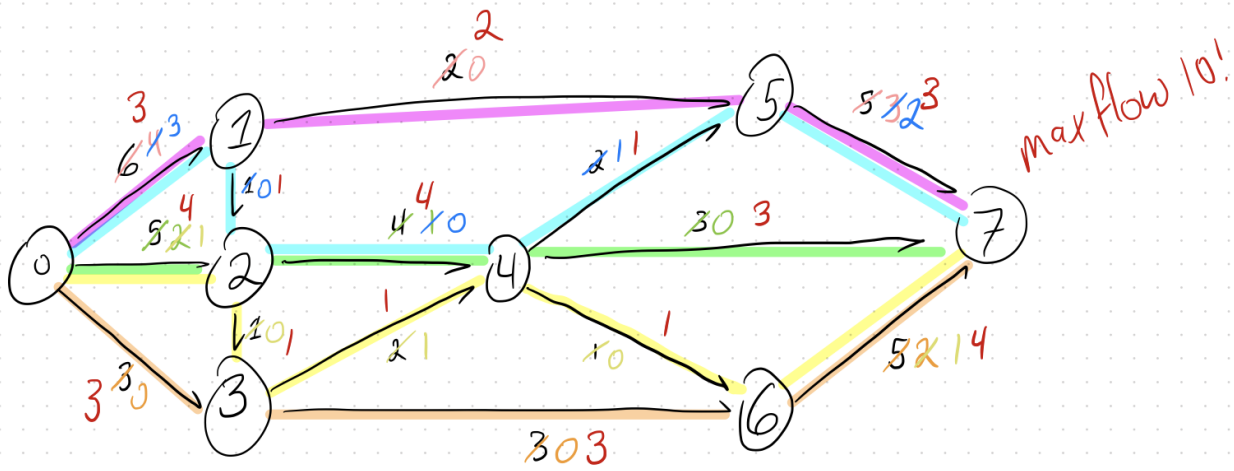
$$
\begin{aligned}
y_1 + 5y_2 &= 11 \\
y_1 - 6y_2 &= 13
\end{aligned}
$$

Giving that $y_1 = 11\frac{10}{11}, y_2 = -\frac{2}{11}$

Problem 2.

a.

I solve the problem in two ways (just to check myself) and I get two different allocations, but they both result in a maximum flow of 10. The second method uses the code that I wrote for pset 7.



In [66]:
```
edge_mat_1=np.transpose(
    np.array( [
              [0,0,0,1,1,2,2,3,3,4,4,4,5,6],
              [1,2,3,2,5,3,4,4,6,5,6,7,7,7],
              [6,5,3,1,2,1,4,2,3,2,1,3,5,5]]))
```

In [67]:
```
#used to make label names:
  #source: geeksforgeeks.org/python-insert-the-string-at-the-beginning-of-all-items-in-a-list/
def prepend(list, str):

  # Using '% s'
  str += '% s'
  list =  [str % i for i in list]
  return(list)
```

In [68]:
```
import itertools
import pandas

def create_eq_matrix(nvert,edgemat):

  A=np.zeros((nvert,(nvert**2)))  #too large will shrink later,
                                  #but makes for easy indexing
  c=np.zeros(nvert**2) # ditto

  #loop over the edgemat, taking each path as an iterand
  for count, value in enumerate(edgemat[:,0:2]):

    col_num=(nvert)*(value[0])+(value[1]) # n columns from each vertex

    if (value[0]==0): #coming from source
      #Don't have to worry about constraint on sources
      A[value[1],col_num]=1
    elif (value[1]==(nvert-1)): #going to sink
      #dont have to worry about constraint on sink
      A[value[0],col_num]=-1
      c[col_num]=-1#Hey this is an important column, but it in the objective
    else: #intermediate
      A[value[1],col_num]=1
      A[value[0],col_num]=-1

  A=np.append(A,np.array([c]),axis=0) #add the row of cs

  #Name appropriately
  col_names=[] #create vector of names
  for i in itertools.product(np.arange(nvert),repeat=2): #all combos of indexes
```

```
    col_names += [str(i[0])+","+str(i[1])]
  row_names=prepend(np.arange(nvert),"V")
  row_names+="c" #add one final row name

  A=pd.DataFrame(A,index=row_names,columns=col_names)# put things into pd data

  # Remove unused rows and columns
  A=A.loc[(A != 0).any(axis=1), (A != 0).any(axis=0)]

  #separate A and c again
  c=A.iloc[-1:,:]
  A=A.iloc[:-1,:]

  #create easy matrices and vectors
  b_eq=np.zeros(A.shape[0])
  A_leq=np.identity(edgemat.shape[0])
  b_leq=edgemat[:,2]

  #do optimization
  optimal=linprog(c,A_eq=A,A_ub=A_leq,b_eq=b_eq,b_ub=b_leq,method="simplex")
  flows=np.array(np.array([optimal.x]))

  #generateoutput
  summary=pd.DataFrame(flows,index=["flow"],columns=A.columns)
  max="The maximum is: " +str(-optimal.fun)

  #print output and return
  print(max)
  return (summary,A)
```

In [69]:
```
create_eq_matrix(8,edge_mat_1)[0]
```

```
The maximum is: 10.0
```

Out[69]:

	0,1	0,2	0,3	1,2	1,5	2,3	2,4	3,4	3,6	4,5	4,6	4,7	5,7	6,7
flow	3.0	4.0	3.0	1.0	2.0	1.0	4.0	1.0	3.0	2.0	1.0	2.0	4.0	4.0

b.

In [70]:
```
edge_mat_2=np.transpose(
    np.array( [
              [0,0,0,1,1,2,2,3,3,4,4,4,5,6],
              [1,2,3,2,5,3,4,4,6,5,6,7,7,7],
              [6,5,3,1,2,1,4,2,3,2,1,3,5,5],
              [3,4,3,4,3,4,4,3,3,3,3,3,3,3]]))
```

In [71]:
```
# Gabe and I converted **his** maxflow code, but i had a substantial part
# in this min cost implementation

def maxflow(nvert,edgemat,reqflow) :
  nrows = edgemat.shape[0]
  A = np.identity(nrows)
  b = np.zeros((1,nrows))
  for i in range(nrows) :
    b[0,i] = edgemat[i][2]
  b2 = np.zeros((1,nvert-2))
  A3 = np.zeros((1,nrows))
  for i in range(nrows) :
    if edgemat[i][1] == nvert-1:
      A3[0,i] = 1
  A2 = np.zeros([nvert-2,nrows])
  for count, value in enumerate(edgemat[:,0:2]):
    colnum = count
    count = count+1
    if (value[0]==0) :
      A2[value[1]-1,colnum] = 1
    elif (value[1]==(nvert-1)) :
      A2[value[0]-1,colnum] = -1
    else :
      A2[value[1]-1,colnum] = 1
      A2[value[0]-1,colnum] = -1
  c = edgemat[:,3]
  #print(A)
  #print(c)
  reqflow = np.array([[reqflow]])
  print(linprog(c,A_ub=A,b_ub=b,A_eq=np.append(A2,A3,axis=0),b_eq=np.append(b2,reqflow,axis=1),method='simplex')
```

In [72]:
```
maxflow(8,edge_mat_2,8)
```

```
     con: array([0., 0., 0., 0., 0., 0., 0.])
     fun: 78.0
 message: 'Optimization terminated successfully.'
     nit: 28
   slack: array([4., 2., 0., 1., 0., 1., 1., 2., 0., 2., 1., 0., 3., 2.])
  status: 0
 success: True
       x: array([2., 3., 3., 0., 2., 0., 3., 0., 3., 0., 0., 3., 2., 3.])
```

This implies the following optimal flows within minimum cost equal to 78:

$$
\begin{aligned}
(0,1) &= 2 \\
(0,2) &= 3 \\
(0,3) &= 3 \\
(1,2) &= 0 \\
(1,5) &= 2 \\
(2,3) &= 0 \\
(2,4) &= 3 \\
(3,4) &= 0 \\
(3,6) &= 3 \\
(4,5) &= 0 \\
(4,6) &= 0 \\
(4,7) &= 3 \\
(5,7) &= 2 \\
(6,7) &= 3
\end{aligned}
$$

Problem 3.

Note first that this is not yet the standard assignment problem because we may need more than one of each task (where each task is a type of food).

I number the players according to their appearance in the given table, $I = \{1, \ldots, 12\}$. I abbreviate the meal types as $F = \{M, A, De, \&Dr\}$ respectively. Therefore, we have the following set of independent variables, $x_{i,f}$ where i is every element in I and f is every element in f. For example $x_{3,De}$ is the number of desserts that Danny provides.

Of course, each player must provide exactly one food. giving that the sum of each agents provisions must equal one.

A sample constraint would then be:

$$x_{1,M} + x_{1,A} + x_{1,De} + x_{1,Dr} = 1$$

Furthermore, we have the constraints saying that $4, 3, 3, \& \ 2$ of each meal type must be provided giving constranits of the forms:

$$
\begin{aligned}
\sum_{i=1}^{12} x_{i,M} &= 4 \\
\sum_{i=1}^{12} x_{i,A} &= 3 \\
\sum_{i=1}^{12} x_{i,De} &= 3 \\
\sum_{i=1}^{12} x_{i,Dr} &= 2
\end{aligned}
$$

This take care of the constraints, the only thing to mention now is the objective function which is simply to maximize the sum:

$$\sum_{i,f \in I \times F} a_{i,f} x_{i,f}$$

where the $a_{i,f}$ is the preference rankings as given.

I arrange the columns in the following order:

$$x_{1,M} \ x_{1,A} \ x_{1,De} \ x_{1,Dr} \ x_{2,M} \ldots$$

The meal constraints can be represented in the following matrix:

In [73]:
```
ident=np.identity(4)
zero=np.zeros((4,4))
food=np.concatenate((ident,ident,ident,ident,ident,ident,ident,ident,ident,ident,ident,ident),axis=1)
food_b=np.array([4,3,3,2])
print(food)

ones=np.ones((1,4))
zero=np.zeros((1,4))

print("And the individual constraint matrix would be: ")
individual=np.block([[ones,zero,zero,zero,zero,zero,zero,zero,zero,zero,zero,zero],
                     [zero,ones,zero,zero,zero,zero,zero,zero,zero,zero,zero,zero],
                     [zero,zero,ones,zero,zero,zero,zero,zero,zero,zero,zero,zero],
                     [zero,zero,zero,ones,zero,zero,zero,zero,zero,zero,zero,zero],
                     [zero,zero,zero,zero,ones,zero,zero,zero,zero,zero,zero,zero],
                     [zero,zero,zero,zero,zero,ones,zero,zero,zero,zero,zero,zero],
                     [zero,zero,zero,zero,zero,zero,ones,zero,zero,zero,zero,zero],
                     [zero,zero,zero,zero,zero,zero,zero,ones,zero,zero,zero,zero],
                     [zero,zero,zero,zero,zero,zero,zero,zero,ones,zero,zero,zero],
                     [zero,zero,zero,zero,zero,zero,zero,zero,zero,ones,zero,zero],
                     [zero,zero,zero,zero,zero,zero,zero,zero,zero,zero,ones,zero],
                     [zero,zero,zero,zero,zero,zero,zero,zero,zero,zero,zero,ones]
                     ])
individual_b=np.ones(12)

print(individual)

final_matrix=np.append(food,individual,axis=0)
final_b=np.append(food_b,individual_b)




votes = np.array([[1,5,2,7],
                  [2,6,5,6],
                  [4,7,1,6],
                  [3,5,2,2],
                  [4,1,1,4],
                  [2,6,1,4],
                  [7,5,6,5],
                  [7,5,5,3],
                  [4,6,3,6],
                  [6,6,7,2],
                  [1,3,1,5],
                  [3,5,4,5]])
```

```
[[1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0.
  1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0.]
 [0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0.
  0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0.
  0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1.
  0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 1.]]
And the individual constraint matrix would be:
[[1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1.]]
```

In [74]:
```
output=np.reshape(linprog(-np.concatenate(votes),A_eq=final_matrix,b_eq=final_b,method="simplex").x,(12,4))
pd.DataFrame(output,
```

```
                index=["Seth","Joel","Danny","Tobias","George","Dwight","Furkan","Tyrese","Shake","Mike","Ben","Mat
                columns=["Main","App","Dessert","Drink"])
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: OptimizeWarning: A_eq does not appear to be of f
ull row rank. To improve performance, check the problem formulation for redundant equality constraints.
  """Entry point for launching an IPython kernel.
```

Out[74]:

	Main	App	Dessert	Drink
Seth	0.0	0.0	0.0	1.0
Joel	0.0	0.0	1.0	0.0
Danny	0.0	1.0	0.0	0.0
Tobias	0.0	1.0	0.0	0.0
George	1.0	0.0	0.0	0.0
Dwight	0.0	1.0	0.0	0.0
Furkan	1.0	0.0	0.0	0.0
Tyrese	1.0	0.0	0.0	0.0
Shake	1.0	0.0	0.0	0.0
Mike	0.0	0.0	1.0	0.0
Ben	0.0	0.0	0.0	1.0
Matisse	0.0	0.0	1.0	0.0

When the table above is one, that corresponds to that person providing one of that food item. The total sum was 68, meaning every person got the choice they rated at 5.66, which is pretty good!

Problem 4

This problem can be set up very similarly to the previous because they are both more or less generalized transportation problems. In this case let $x_{i,j}$ be the quantity sent from Wi to Fj. The festivals need their respective quantities of ice cream giving eqs of the following form:

$$\forall j \in \{1,2,3,4,5\}, \sum_{i=1}^{4} x_{ij} \geq d_j$$

Where d_j is the given required ice cream at festival j. However these constraints are not ideal for the simplex form, so multiplying by -1 gives constraints:

$$\forall j \in \{1,2,3,4,5\}, \sum_{i=1}^{4} -x_{ij} \leq -d_j$$

And then we have the warehouse constraints of the following form

$$\forall i \in \{1,2,3,4\}, \sum_{j=1}^{5} x_{ij} \leq s_i$$

Where s_i is the quantity available at each warehouse.

The objective function is to minimize the sum of the costs, c_{ij}.

We now how constraints and an obj function that lin prog can deal with. I order the variables in the following way, and then proceed to solve the problem:

$$x_{11}\ x_{12} \ldots$$

In [75]:

```
ident=np.identity(5)
festivals=-1*np.concatenate((ident,ident,ident,ident),axis=1)
festivals_b=-1*np.array([2,3,3,6,5])


ones=np.ones((1,5))
zero=np.zeros((1,5))

warehouse=np.block([[ones,zero,zero,zero],
                    [zero,ones,zero,zero],
                    [zero,zero,ones,zero],
```

```
                    [zero,zero,zero,ones]])

warehouse_b=np.array([8,6,4,3])

full_matrix=np.concatenate((festivals,warehouse),axis=0)
full_b=np.concatenate((festivals_b,warehouse_b),axis=0)

obj=np.array([8,6,9,15,13,12, 9, 5, 11.5, 8,10, 12.5, 11, 6, 4,5.5, 8, 12, 6, 9])

output=linprog(obj,A_ub=full_matrix,b_ub=full_b,method="simplex")
print("The min cost is: "+ str(output.fun))
pd.DataFrame(np.reshape(output.x,(4,5)),index=["W1","W2","W3","W4"],columns=["F1","F2","F3","F4","F5"])
```

```
The min cost is: 125.0
```

Out[75]:

	F1	F2	F3	F4	F5
W1	2.0	3.0	1.0	0.0	0.0
W2	0.0	0.0	2.0	0.0	4.0
W3	0.0	0.0	0.0	3.0	1.0
W4	0.0	0.0	0.0	3.0	0.0

The table above summarizes how much each warehouse should transport to each to festival.

Problem 5.

The indepdendent variables are simply the number of people assigned to each 8 hr shift. Number the shifts as they appear in the table of wages, 1 to 12.

Note that each shift covers 4 "times of day" but can wrap around over midnight.

Further note that we need *at least* the specified number of workers at each "time of day" so we have $\geq$ constraints that I conver to leq by multiplying by -1. At first I said oh i'll just make them all equality constraints, but I soon realized that it was impossible to higher a number of workers that exactly satisfies each constraint. This is because matrix is not full row rank!

I now write a nifty bit of code to generate the scheduling matrix:

In [76]:
```
a=np.zeros((12,12))
for i in range(12):
  col_1=i # starting column
  col_2=((i+3)%12)+1 #plus one bc vector indexing is non-inlcusive
  if (col_1<col_2): # if no wrap around
    a[col_1:col_2,i]=1 # continguous set of ones
  else: #must be wrap around
    a[col_1:12,i]=1 # go from starting point to end of matrix
    a[0:col_2,i]=1 # wrap around
a=-1*a
print(a)
```

```
[[-1. -0. -0. -0. -0. -0. -0. -0. -0. -1. -1. -1.]
 [-1. -1. -0. -0. -0. -0. -0. -0. -0. -0. -1. -1.]
 [-1. -1. -1. -0. -0. -0. -0. -0. -0. -0. -0. -1.]
 [-1. -1. -1. -1. -0. -0. -0. -0. -0. -0. -0. -0.]
 [-0. -1. -1. -1. -1. -0. -0. -0. -0. -0. -0. -0.]
 [-0. -0. -1. -1. -1. -1. -0. -0. -0. -0. -0. -0.]
 [-0. -0. -0. -1. -1. -1. -1. -0. -0. -0. -0. -0.]
 [-0. -0. -0. -0. -1. -1. -1. -1. -0. -0. -0. -0.]
 [-0. -0. -0. -0. -0. -1. -1. -1. -1. -0. -0. -0.]
 [-0. -0. -0. -0. -0. -0. -1. -1. -1. -1. -0. -0.]
 [-0. -0. -0. -0. -0. -0. -0. -1. -1. -1. -1. -0.]
 [-0. -0. -0. -0. -0. -0. -0. -0. -1. -1. -1. -1.]]
```

With that it is now as simple as adding the b vector and minimizing the cost in the following way:

In [77]:
```
b=-1*np.array([100,65,65,95,200,335,365,285,180,135,120,110])

cost=np.array([20,20,20,18,16,16,18,18,18,20,20,20])

linprog(cost,A_ub=a,b_ub=b,method="simplex")
sum(linprog(cost,A_ub=a,b_ub=b,method="simplex").x)
```

Out[77]:
```
550.0
```

The minimum cost is 9460, or with a minimum 550 workers, a minimum average pay of $\$17.2$. We hire 65 of midnight to 8, 30 6-2, 240 8-4, 80 10-6, 15 12pm-8pm, 10 2pm-10pm, 75 4pm-12am, 35 6pm-2am, and zero of all other shifts.