Branch-and Bound to solve Integer Linear Programming problems

Authors: HE Chenchen

Import the libraries

In [1]:

```
import numpy as np
from scipy.optimize import linprog
import math
```

Tuning parameters (if necessary)

Function "isInteger(x)"

In [2]:

```
def isInteger( x ):
    """
    As for a vector x,check if it contains only integer values:
    return the boolean value True and the None, i.e., [True,None] when the values are all integer
    return False and the index of the noninteger value in x, i.e., [False,index]

    """
    xx=np.array(x)
    dist=np.array(abs(np.rint(xx)-xx))
    for xx in dist:
        if float(xx).is_integer()== False:
            dist[dist==0]=np.nan
            return (False,np.nanargmin(dist))
    return (True,None)
```

Class "Node"

In [3]:

```
class Node:
    """
    This class models a node in the branch and bound algorithm.
    x: the solution in node
    z: the value of node
    status: the status of the node, if this node is availabe

    """
    def __init__ (self,x,z,bounds,status):
        self.x = x
        self.z = z
        self.bounds = bounds
        self.status = status
```

Definition of the LP problem

Exercise 8.1

Let us consider the following ILP problem

$max_{(x1,x2) \in R^2} \quad z = 10x_1 + 20x_2$

s.t. $5x_1 + 8x_2 \leq 60$

$x_1 \leq 8$

$x_2 \leq 4$

$x_1, x_2 \in N$

Solve the ILP with Dakin's method. In each node, separate on the fractional variable which is the closest to an integer.

Exercise 8.2

Let us consider the following ILP problem

$max_{(x1,x2,x3,x4) \in R^4} \quad z = 2x_1 + 3x_2 + x_3 + 2x_4$

s.t. $5x_1 + 2x_2 + x_3 + x_4 \leq 15$

$2x_1 + 6x_2 + 10x_3 + 8x_4 \leq 60$

$x_1 + x_2 + x_3 + x_4 \leq 8$

$2x_1 + 2x_2 + 3x_3 + 3x_4 \leq 16$

$x_1 \leq 3$

$x_2 \leq 7$

$x_3 \leq 5$

$x_4 \leq 5$

$x_1, x_2, x_3, x_4 \in N$

Solve the ILP with Dakin's method. In each node, separate on the fractional variable which is the closest to an integer.

In [4]:

```
# Definition of the integer LP problem

#==========assignment 8.1 ===================

c = np.array([10,20])
A = np.array([[5,8],[1,0],[0,1]])
b = np.array([60,8,4])


#=========assignment 8.2 ===================
'''
c = np.array([2,3,1,2])
A = np.array([[5,2,1,1],[2,6,10,8],[1,1,1,1],[2,2,3,3],[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]])
b = np.array([15,60,8,16,3,7,5,5])
'''
A_eq = None
b_eq = None
n=len(c)

# two ways to initialize bounds automatically

#bounds=np.full((n,2),(0,None))
bounds = [[0, None] for _ in range(n)]
```

Initialization of the branch-and-bound algorithm

In [5]:

```
"""
Initialization of the branch-and-bound algorithm

we have three main input parameters, i.e., node, bestnode, iteration
node: the node we need to proceed
bestnode: is used to store the best solution of the branch-and-bound algorithm
iteration: the iteration of processing node

"""
# Initializing the node
init_res = linprog(-c,A,b,A_eq,b_eq)
x=init_res.x
z=init_res.fun
status=init_res.status
node = Node(x,z,bounds,status)

# Initialize the bestnode
bstnode_status=0
bestnode = Node([0,0],0,bounds,bstnode_status)

# Initialize the iteration
iteration=0
```

Branch-and-Bound recursive processing

Dakin's method:

If $T = \emptyset$:

- Terminate the algorithm. Give the best found integer solution and its profit z (if no solution, the ILP is not feasible)

If $T \neq \emptyset$:

- If $U(S_i) > z$ and the solution only contains integers, it is the best solution found since the begnning: update z

- If $U(S_i) > z$ but the solution contains at least one non-integer, choose a functional variable x_j^* and split S_i: and one child with the additional constraint $x_j \leq [x_j^*]$ and the other one with $x_j \geq [x_j^*]$. Process the two children nodes and insert them into T

- If $U(S_i) \leq z$, discard S_i (prunning): it can not improve z

In [6]:

```
def branch_bound(node,bestnode,iteration):
    """
    We implement the Dakin's method step by step

    <node> denotes T
    <bestnode> denotes the best node with integer solution and best profit, and the bestnode.z stanc
    <iteration> denotes the iteration

    """
    # decide the node is available or not
    if node.status != 0:
        # terminate the algorithm, return the best found integer solution and its profit z
        return (bestnode,iteration)

    # decide update the best solution or not
    if node.z < bestnode.z:
        (isIntegerOrNot,splitIndex)=isInteger(node.x)

        # decide if the solution is integer
        if isIntegerOrNot == True:
            # update the bestnode
            bestnode=node
            # return the bestnode
            return (bestnode,iteration)
        else: # split the node to find the bestnode

            # choose a functional variable to split node
            spValue = node.x[splitIndex]
            # one child with conditional constraints x<=lowbound
            lowbound = np.floor(spValue)
            boundsL=np.array(node.bounds)
            boundsL[splitIndex]=[0,lowbound]
            # process the child node and insert them into the <node> to find the bestnode
            resL=linprog(-c,A,b,A_eq,b_eq,boundsL)
            nodeL = Node(resL.x, resL.fun, boundsL, resL.status)
            iteration=iteration+1
            print("Iteration number is: "+str(iteration)+ "\nProcessed node: Solution x:{0}, Value:
            if nodeL.z <= bestnode.z:
                # start the recursive processing
                (bestnode,iteration)=branch_bound(nodeL,bestnode,iteration)
            else:
                # one child with conditional constraints x>=upbound
                upbound = np.ceil(spValue)
                boundsR=np.array(node.bounds)
                boundsR[splitIndex]=[upbound,None]
                # process the child node and insert them into the <node> to find the bestnode
                resR=linprog(-c,A,b,A_eq,b_eq,boundsR)
                nodeR= Node(resR.x, resR.fun, boundsR, resR.status)
                iteration=iteration+1
                print("Iteration number is: "+str(iteration)+ "\nProcessed node: Solution x:{0}, Va
                if nodeR.z <= bestnode.z:
                    # start the recursive processing
                    (bestnode,iteration)=branch_bound(nodeR,bestnode,iteration)
    else:
        # return the bestnode
        return(bestnode,iteration)
    # return the bestnode
    return (bestnode,iteration)
```

Final messages to the users

In [8]:

```
#print("Original node: x: {0}, Value:{1}, Bounds:{2}".format(node.x,node.z,node.bounds))
(bestnode,iteration)=branch_bound(node,bestnode,iteration)
print("\nAn integer solution is found by improvement!\n")
#print("Optimal solution x:{0}, Value:{1}, Bounds:{2}".format(bestnode.x,-bestnode.z,bestnode.bounds
```

```
Iteration number is: 2
Processed node: Solution x:[5. 4.], Value:130.0, Bounds:[[0 5.0]
 [0 None]]
Split on x_1

An integer solution is found by improvement!
```

In [ ]:

```

```