Branch-and Bound to solve Integer Linear Programming problems

Authors: Sherly Sherly and Alessio Molinari

Import the libraries

In [1]:

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

Integer Linear Programming

Problem formulation:

$$\begin{aligned} \max \quad & c^T x \\ \text{s.t.} \quad & Ax = b \\ & x \geq 0 \\ & x_j \in \mathbb{Z}, \forall j = 1, .., n \end{aligned}$$

where A is a mxn matrix
b an m-dimensional column vector
c is an n-dimensional row vector
x is an n-dimensional column vector of variables

In this notebook, we will be implementing the branch-and-bound method to solving Integer Linear Programming (ILP) problems. The variable values A, b and c used in following sections of the notebook refers to the values as presented in the problem formulation above.

Branch and Bound

General Algorithm to solving Integer Linear Programming with Branch and Bound based on the problem formulation:

1. Initialize the initial lower bound to $-\infty$
2. Solve Linear Programming relaxation for current node
   - If solution is below lower bound, branch is pruned
   - If solution is integer, replace lower bound
3. Create two branched problems by adding constraints to original problem.
   - Select integer variable with fractional LP solution which is closest to an integer.
   - Add integer constraints to the original LP. Constraints are generated by upper-bounding and lower-bounding the fractional solution. i.e. if $x = 2.1$, add constraints $x \leq 2$ and $x \geq 3$ respectively into each of the two branches.
4. Repeat step 1-2 until no branching occurs, return optimal solution.

Terminologies:

- Bound: The maximum/minimum boundary value for the maximization/minimization problem.

In order to solve the Integer Linear Programming problems, we define several helper functions and classes in the sections below.

1. `isInteger(x)` function to check the branching criteria.
2. `Node` class to store the information at each node and perform node operations.
3. `generate_node(args)` function to solve LP problem with `scipy.optimize.linprog` and store the data into a Node
4. `BranchAndBound` class which stores the global variables within the class and perform recursive branching.

Function "isInteger(x)"

This function will take in a vector, checks for its values and returns (isIntegerOnly, indexOfNonInteger) pair.

- If it contains only integer values, it returns (True, None).
- If it contains at least one non-integer value it returns (False, index) where index refers to the position of the value of non-integer which is the closest to an integer.

In [2]:

```python
def isInteger( x ):
    """
    Takes in a vector list and checks if the values are integers.
    Returns (True, None) if it is all integers
    Returns (False, index of value closest to integer) if there are non-integers
    """
    # retrieve all the difference between the float and the nearest integer
    diffInt = np.array([abs(v - np.rint(v)) for v in x])

    for v in diffInt:
        if float(v).is_integer() == False:
            diffInt[diffInt == 0] = np.nan
            return (False, np.nanargmin(diffInt))

    return (True, None)
```

In [3]:

```python
# Test cases
print(isInteger([1, 2.1, 3.95, 5.9]))
print(isInteger([1, 2, 3.0, 3]))
```

```
(False, 2)
(True, None)
```

Class "Node"

The class Node models after the node in the branch-and-bound algorithm. It can be initialized with the values of:

- x : the optimized solution
- profit : the maximum value
- bounds : the x bounds for the optimization problem
- feasible : the feasibility of the solution
- split_var : the parent branching variable

The function `update_bounds()` takes in the new criterias and generate a new bound for the branching process.
Example:

- Original bounds: [(0, None), (0, None)]
- Original x values: [2.1, 1.2]
- idx = 0 is to update index 0 bounds (note: list index 0 corresponds to variable x_1)
- For methods:
  - "ceil" : ceiling the values of x_1 i.e. 3.
    Update the bounds for x_1 to include the new constraints $x_1 \geq 3$ i.e. bounds = [(3, None), (0, None)]
  - "floor" : flooring the values of x_1 i.e. 2.
    Update the bounds for x_1 to include the new constraints $x_1 \geq 3$ i.e. bounds = [(0, 2), (0, None)]

The function `printData()` is a helper function to show the information in the node.

In [4]:

```
import copy

class Node:

    def __init__(self, x, profit, bounds, feasible=None, split_var=None):
        self.x = x                  # value of x
        self.profit = profit        # optimized value
        self.bounds = bounds
        self.split_var = split_var # index of the parent branching variable
        self.feasible = feasible

    def update_bounds(self, idx, method):
        """
        This function takes in an index for the branching variable and
        generate new bounds based on the existing bounds accordingly.
        This is called whenever branching happens where new constraints
        are being added. Bounds are initialized as tuples to prevent
        accidental updates on the values.

        Arguments:
          - idx    (int) : index of the branching variable
          - method (str) : method takes in either "ceil" or "floor"
                           which corresponds to the type of branching

        Returns:
          - List of tuples of bounds with new constraints.
        """
        # to prevent updating of original bounds
        bounds = copy.deepcopy(self.bounds)

        if method == "floor":
            _x = np.floor(self.x[idx])
            toUpdate = list(bounds[idx])
            toUpdate[1] = _x
            bounds[idx] = tuple(toUpdate)

        elif method == "ceil":
            _x = np.ceil(self.x[idx])
            toUpdate = list(bounds[idx])
            toUpdate[0] = _x
            bounds[idx] = tuple(toUpdate)
        else:
            return None

        return bounds

    def printData(self):
        _str = "Processed node : Solution x : {x}, Value : {profit}, Bounds : {b
ounds}"
        print(_str.format(
            x = self.x,
            profit = self.profit,
            bounds = self.bounds))

        if not self.split_var == None:
            print("Node is a split from x_{} in previous node".format(self.split
_var+1))
```

The function `generate_node` is a helper function to solve the linear programming problem and generate a node which is a `Node` class object for the solution.

It wraps around the `scipy.optimize.linprog` function that solves a minimization problem by default into a function that performs maximization on the linear programming problem as defined by our problem formulation.

In [5]:

```
def generate_node(c, A, b, bounds=None, **kwargs):
    """
    By default, `scipy.optimize.linprog` performs minimization.
    This function takes care of maximization problem.
    It inverts the values of c before feeding it into `linprog` to perform
    maximization.
    It checks for the feasibility of the linear programming solution
    and generates a Node according to the feasiblity.
    Arguments:
        - c       (np.array([])) : coeffs of the maximization problem
        - A (np.array([[], []])) : coeffs of the constraints
        - b       (np.array([])) : constants of the constraints
        - bounds      ([(), ()]) : limits for the LP (optional), if not
                                   initialized, it is set to be list of (0, Non
e)
        - kwargs                 : additional args for Node()
    Return:
        Initialized Node object for the optimization result.
    """
    if bounds is None:
        bounds = [(0, None) for _ in range(len(c))]

    lp_sol = linprog(c*-1, A_ub=A, b_ub=b,
                     bounds=bounds)

    # success is True if the algorithm succeeded in finding an optimal solution
    if lp_sol.success is False:
        return Node(None, -np.inf, None, False, **kwargs)

    # The return values of the `scipy.optimize.linprog` are for minimization
    # and hence we multiply the results by -1
    return Node(
        x        = lp_sol.x,
        profit   = lp_sol.fun*-1,
        # we multiply the optimal value by -1 to invert it to max
        bounds   = bounds,
        feasible = True,
        **kwargs
    )
```

Initialization of variables for the problem

Initialize the values A, b, and c according to the problem formulation. Bounds are initialized according to the constraints of the variables.

Data types:

- A: `np.array([])`
- b: `np.array([])`
- c: `np.array([])`
- bounds: `[(), ()]` list of tuples

Note: based on the problem formulation, we are solving a maximization problem. Initialize the parameters accordingly.
Hint: $\min c^T x \Leftrightarrow \max - c^T x$

To visualize how this is initialized, let us consider the following ILP problem from Exercise 8.1:

$$\begin{aligned} \max_{(x_1,x_2)\in\mathbb{R}} \quad & z = 10x_1 + 20x_2 \\ \text{s.t.} \quad & 5x_1 + 8x_2 \leq 60 \\ & x_1 \leq 8 \\ & x_2 \leq 4 \\ & x_1, x_2 \in \mathbb{N} \end{aligned}$$

In [6]:

```
c = np.array([10, 20])
A = np.array([5, 8])
b = np.array([60])
# bounds has to be of type [()] i.e. list of tuples
bounds = [(0, 8), (0, 4)]

S0 = generate_node(c, A, b, bounds=bounds)
```

Branch-and-Bound recursive processing

The class `BranchAndBound` contains methods to solve the branch and bound problem recursively.
It is initialized with parameters A, b, c and $S0$. The class keeps track of the iteration counts, the list of nodes, the best solution and the best node which is required in the recursive branching.

<u>Dakin's Method</u>
For each node:

- If $\text{profit} \leq \text{optimal}$, prune the node as it can not improve z.
- If $\text{profit} > \text{optimal}$ and the solution only contains integers, it is the best solution found since the beginning: update optimal.
- If $\text{profit} > \text{optimal}$ and the solution contains at least one non-integer, choose a fractional variable x_i which is closest to an integer and branch from the node into two with additional constraint $x \geq \lceil x_i \rceil$ and $x \leq \lfloor x_i \rfloor$ respectively.

In [7]:

```
# Implementation of Dakin's method

class BranchAndBound:

    def __init__(self, c, A, b, S0):
        self.c = c
        self.A = A
        self.b = b
        self.S0 = S0

        # List of nodes
        self.T = [S0]
        self.countIter = 0
        self.bestSol = -np.inf   # initialization
        self.bestNode = None

    def branch(self, node=None):
        if not node and self.countIter == 0:
            node = self.S0

        self.countIter += 1
        print("=========================================================")
        print("Iteration number {}".format(self.countIter))
        node.printData()

        # if node.profit <= bestSol, discard node i.e. prune as it cannot improv
e
        if node.profit <= self.bestSol:
            print("Node is pruned as it's profit is less than or equal",
                  "to the best available.")
            return node

        # from this stage onwards, we know that node.profit > bestSol
        isInt, idx = isInteger(node.x)
        if isInt:
            self.bestSol = node.profit
            self.bestNode = node
            print("An integer solution is found with an improvement!")
            return node


        # At least one of the values of Xi is non-integer, branch the node.
        # Branching is performed on Xi which is closest to an integer
        print("Proceed to split on x_{}".format(idx+1))

        # For each branch, add in the new constraints into the bounds
        # and generate the new node. If the node is feasible, proceed to
        # perform another branching if conditions are satisfied by
        # calling this function recursively.
        boundsFloor = node.update_bounds(idx=idx, method="floor")

        if boundsFloor is not None:
            nodeFloor = generate_node(c = self.c, A = self.A, b = self.b,
                                      bounds = boundsFloor, split_var = idx)
            self.T.append(nodeFloor)
            if nodeFloor.feasible:
                self.branch(nodeFloor)
```

```
        boundsCeil = node.update_bounds(idx=idx, method="ceil")

        if boundsCeil is not None:
            nodeCeil = generate_node(c = self.c, A = self.A, b = self.b,
                                     bounds = boundsCeil, split_var = idx)
            self.T.append(nodeCeil)
            if nodeCeil.feasible:
                self.branch(nodeCeil)
```

In [8]:

```
# Sample Solution

branchBound = BranchAndBound(c, A, b, S0)
branchBound.branch()
print("=======================================================")
print("Integer solution to the problem is {} with profit {}".format(
    branchBound.bestNode.x, branchBound.bestSol))
```

```
=======================================================
Iteration number 1
Processed node : Solution x : [5.6 4. ], Value : 136.0, Bounds :
[(0, 8), (0, 4)]
Proceed to split on x_1
=======================================================
Iteration number 2
Processed node : Solution x : [5. 4.], Value : 130.0, Bounds : [(0,
5.0), (0, 4)]
Node is a split from x_1 in previous node
An integer solution is found with an improvement!
=======================================================
Iteration number 3
Processed node : Solution x : [6.   3.75], Value : 135.0, Bounds :
[(6.0, 8), (0, 4)]
Node is a split from x_1 in previous node
Proceed to split on x_2
=======================================================
Iteration number 4
Processed node : Solution x : [7.2 3. ], Value : 132.0, Bounds :
[(6.0, 8), (0, 3.0)]
Node is a split from x_2 in previous node
Proceed to split on x_1
=======================================================
Iteration number 5
Processed node : Solution x : [7. 3.], Value : 130.0, Bounds : [(6.
0, 7.0), (0, 3.0)]
Node is a split from x_1 in previous node
Node is pruned as it's profit is less than or equal to the best avai
lable.
=======================================================
Iteration number 6
Processed node : Solution x : [8.  2.5], Value : 130.0, Bounds :
[(8.0, 8), (0, 3.0)]
Node is a split from x_1 in previous node
Node is pruned as it's profit is less than or equal to the best avai
lable.
=======================================================
Integer solution to the problem is [5. 4.] with profit 130.0
```

Final messages to the users

Steps to solving Integer Linear Programming problem with Branch and Bound:

1. Initialize A, b, c and bound for the maximization problem.
   Recall: $\min c^T x \Leftrightarrow \max -c^T x$
2. Generate initial node $S0$ by calling the `generate_node()` with the variables
3. Initialize `BranchAndBound` class with c, A, b and $S0$
4. Call the `branch` method of the `BranchAndBound` class to generate the optimal solution.
5. Retrieve the best solution from `BranchAndBound` class attributes `bestNode.x` and `branchBound.bestSol`

Exercise 8.1

Let us consider the following ILP problem :

$$
\begin{aligned}
\max_{(x_1,x_2)\in\mathbb{R}} \quad & z = 10x_1 + 20x_2 \\
\text{s.t.} \quad & 5x_1 + 8x_2 \leq 60 \\
& x_1 \leq 8 \\
& x_2 \leq 4 \\
& x_1, x_2 \in \mathbb{N}
\end{aligned}
$$

Solve the ILP with Dakin's method.

In [9]:

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

S0 = generate_node(c, A, b, bounds=bounds)

branchBound = BranchAndBound(c, A, b, S0)
branchBound.branch()
print("=======================================================")
print("Integer solution to the problem is {} with profit {}".format(
    branchBound.bestNode.x, branchBound.bestSol))
```

```
=======================================================
Iteration number 1
Processed node : Solution x : [5.6 4. ], Value : 136.0, Bounds :
[(0, 8), (0, 4)]
Proceed to split on x_1
=======================================================
Iteration number 2
Processed node : Solution x : [5. 4.], Value : 130.0, Bounds : [(0,
5.0), (0, 4)]
Node is a split from x_1 in previous node
An integer solution is found with an improvement!
=======================================================
Iteration number 3
Processed node : Solution x : [6.   3.75], Value : 135.0, Bounds :
[(6.0, 8), (0, 4)]
Node is a split from x_1 in previous node
Proceed to split on x_2
=======================================================
Iteration number 4
Processed node : Solution x : [7.2 3. ], Value : 132.0, Bounds :
[(6.0, 8), (0, 3.0)]
Node is a split from x_2 in previous node
Proceed to split on x_1
=======================================================
Iteration number 5
Processed node : Solution x : [7. 3.], Value : 130.0, Bounds : [(6.
0, 7.0), (0, 3.0)]
Node is a split from x_1 in previous node
Node is pruned as it's profit is less than or equal to the best avai
lable.
=======================================================
Iteration number 6
Processed node : Solution x : [8.  2.5], Value : 130.0, Bounds :
[(8.0, 8), (0, 3.0)]
Node is a split from x_1 in previous node
Node is pruned as it's profit is less than or equal to the best avai
lable.
=======================================================
Integer solution to the problem is [5. 4.] with profit 130.0
```

Exercise 8.2

Let us consider the following ILP problem :

$$\begin{aligned}
\max_{(x_1,x_2)\in\mathbb{R}} \quad & z = 2x_1 + 3x_2 + x_3 + 2x_4 \\
\text{s. t.} \quad & 5x_1 + 2x_2 + x_3 + x_4 \le 15 \\
& 2x_1 + 6x_2 + 10x_3 + 8x_4 \le 60 \\
& x_1 + x_2 + x_3 + x_4 \le 8 \\
& 2x_1 + 2x_2 + 3x_3 + 3x_4 \le 16 \\
& x_1 \le 3 \\
& x_2 \le 7 \\
& x_3 \le 5 \\
& x_4 \le 5 \\
& x_1, x_2, x_3, x_4 \in \mathbb{N}
\end{aligned}$$

Based on the bounds of the integer variables, how many distinct vectors x should be examined to solve the ILP by complete enumeration? Solve it with Dakin's method.

In [10]:

```
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]])
b = np.array([15, 60, 8, 16])
bounds = [(0, 3), (0, 7), (0, 5), (0, 5)]

S0 = generate_node(c, A, b, bounds=bounds)

branchBound = BranchAndBound(c, A, b, S0)
branchBound.branch()
print("======================================================")
print("Integer solution to the problem is {} with profit {}".format(
    branchBound.bestNode.x, branchBound.bestSol))
```

```
======================================================
Iteration number 1
Processed node : Solution x : [0.07692308 7.         0.         0.61
538462], Value : 22.384615384615383, Bounds : [(0, 3), (0, 7), (0,
5), (0, 5)]
Proceed to split on x_1
======================================================
Iteration number 2
Processed node : Solution x : [0.         7.         0.         0.66
666667], Value : 22.333333333333332, Bounds : [(0, 0.0), (0, 7), (0,
5), (0, 5)]
Node is a split from x_1 in previous node
Proceed to split on x_4
======================================================
Iteration number 3
Processed node : Solution x : [0.         7.         0.66666667 0.
      ], Value : 21.666666666666668, Bounds : [(0, 0.0), (0, 7), (0,
5), (0, 0.0)]
Node is a split from x_4 in previous node
Proceed to split on x_3
======================================================
Iteration number 4
Processed node : Solution x : [0. 7. 0. 0.], Value : 21.0, Bounds :
[(0, 0.0), (0, 7), (0, 0.0), (0, 0.0)]
Node is a split from x_3 in previous node
An integer solution is found with an improvement!
======================================================
Iteration number 5
Processed node : Solution x : [0.  6.5 1.  0. ], Value : 20.5, Bound
s : [(0, 0.0), (0, 7), (1.0, 5), (0, 0.0)]
Node is a split from x_3 in previous node
Node is pruned as it's profit is less than or equal to the best avai
lable.
======================================================
Iteration number 6
Processed node : Solution x : [0.  6.5 0.  1. ], Value : 21.5, Bound
s : [(0, 0.0), (0, 7), (0, 5), (1.0, 5)]
Node is a split from x_4 in previous node
Proceed to split on x_2
======================================================
Iteration number 7
Processed node : Solution x : [0.         6.         0.         1.33
333333], Value : 20.666666666666668, Bounds : [(0, 0.0), (0, 6.0),
(0, 5), (1.0, 5)]
Node is a split from x_2 in previous node
Node is pruned as it's profit is less than or equal to the best avai
lable.
======================================================
Iteration number 8
Processed node : Solution x : [1. 4. 0. 2.], Value : 18.0, Bounds :
[(1.0, 3), (0, 7), (0, 5), (0, 5)]
Node is a split from x_1 in previous node
Node is pruned as it's profit is less than or equal to the best avai
lable.
======================================================
Integer solution to the problem is [0. 7. 0. 0.] with profit 21.0
```

In [11]:

```
c = np.array([3, 4])
A = np.array([[0.4, 1], [0.4, -0.4]])
b = np.array([3, 1])

S0 = generate_node(c, A, b)

branchBound = BranchAndBound(c, A, b, S0)
branchBound.branch()
print("======================================================")
print("Integer solution to the problem is {} with profit {}".format(
    branchBound.bestNode.x, branchBound.bestSol))
```

```
======================================================
Iteration number 1
Processed node : Solution x : [3.92857143 1.42857143], Value : 17.5,
Bounds : [(0, None), (0, None)]
Proceed to split on x_1
======================================================
Iteration number 2
Processed node : Solution x : [3.  1.8], Value : 16.2, Bounds : [(0,
3.0), (0, None)]
Node is a split from x_1 in previous node
Proceed to split on x_2
======================================================
Iteration number 3
Processed node : Solution x : [3. 1.], Value : 13.0, Bounds : [(0,
3.0), (0, 1.0)]
Node is a split from x_2 in previous node
An integer solution is found with an improvement!
======================================================
Iteration number 4
Processed node : Solution x : [2.5 2. ], Value : 15.5, Bounds : [(0,
3.0), (2.0, None)]
Node is a split from x_2 in previous node
Proceed to split on x_1
======================================================
Iteration number 5
Processed node : Solution x : [2.  2.2], Value : 14.8, Bounds : [(0,
2.0), (2.0, None)]
Node is a split from x_1 in previous node
Proceed to split on x_2
======================================================
Iteration number 6
Processed node : Solution x : [2. 2.], Value : 14.0, Bounds : [(0,
2.0), (2.0, 2.0)]
Node is a split from x_2 in previous node
An integer solution is found with an improvement!
======================================================
Iteration number 7
Processed node : Solution x : [0. 3.], Value : 12.0, Bounds : [(0,
2.0), (3.0, None)]
Node is a split from x_2 in previous node
Node is pruned as it's profit is less than or equal to the best avai
lable.
======================================================
Integer solution to the problem is [2. 2.] with profit 14.0
```