week05-01–branch-and-bound

February 11, 2025

1 George McNinch Math 87 - Spring 2025

2 Integer programming via Branch & Bound

3 Week 5

3.1 Integer programming: summary of some issue(s)

As an example, consider the linear program:

`maximize` $f(x_1, x_2) = x_1 + 5x_2$; i.e. $\mathbf{c} \cdot \mathbf{x}$ where $\mathbf{c} = \begin{bmatrix} 1 & 5 \end{bmatrix}$.

such that $A\mathbf{x} = \begin{bmatrix} 1 & 10 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} 20 \\ 2 \end{bmatrix}$ and $\mathbf{x} \ge \mathbf{0}$.

Let's find the optimal solution $\mathbf{x} \in \mathbb{R}^2$, and the optimal *integral* solution $\mathbf{x}$ with $x_1, x_2 \in \mathbb{Z}$.

We'll start by solving the *relaxed* problem, where the integrality condition is ignored:

[1]:
```
from scipy.optimize import linprog
import numpy as np

A = np.array([[1,10],[1,0]])
b = np.array([20,2])
c = np.array([1,5])

result=linprog((-1)*c,A_ub = A, b_ub = b)
print(f"result = {result.x}\nmaxvalue = {(-1)*result.fun:.2f}")
```

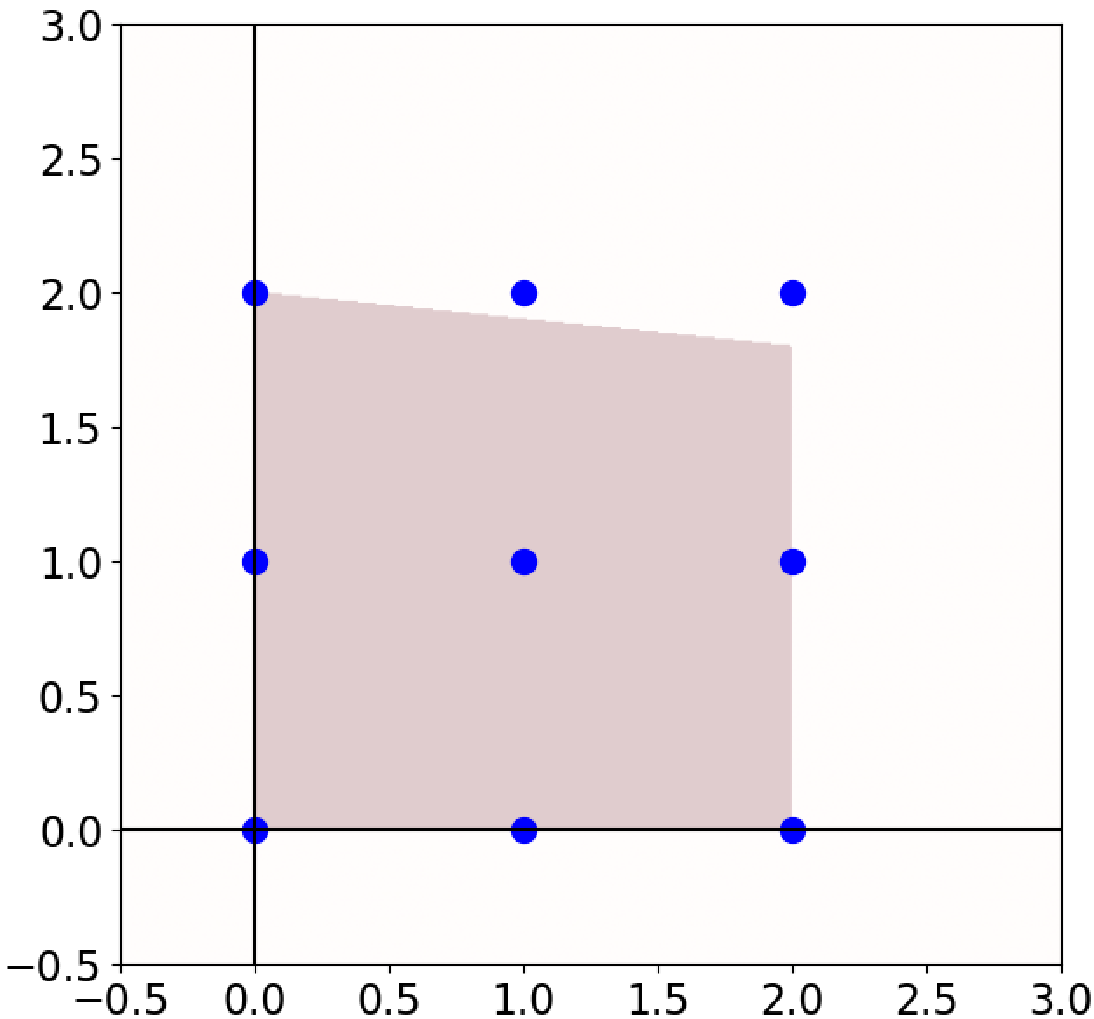
```
result = [2.  1.8]
maxvalue = 11.00
```

This calculution shows that an optimal solution with no integer constraint is $\mathbf{x} = \begin{bmatrix} 2 \\ 1.8 \end{bmatrix}$ and that the optimal value is roughly 11.

Let's make an image of the feasible set:

[2]:
```
%matplotlib notebook
%matplotlib inline
import matplotlib.pyplot as plt
```

```python
import itertools

plt.rcParams.update({'font.size': 17})

# plot the feasible region
d = np.linspace(-.5,3,500)
X,Y = np.meshgrid(d,d)

def vector_le(b,c):
    return np.logical_and.reduce(b<=c)

@np.vectorize
def feasible(x,y):
    p=np.array([x,y])
    if vector_le(A@p,b) and vector_le(np.zeros(2),p):
        return 1.0
    else:
        return 0.0

Z=feasible(X,Y)

fig,ax = plt.subplots(figsize=(7,7))
ax.axhline(y=0, color = "black")
ax.axvline(x=0, color = "black")

# draw the region defind by x >= 0 and Ax <= b.
ax.imshow(Z,
          extent=(X.min(),X.max(),Y.min(),Y.max()),
          origin="lower",
          cmap="Reds",
          alpha = 0.2)

def dot(x,y):
    return ax.scatter(x,y,s=100,color="blue")

# draw the integer points
for i,j in itertools.product(range(3),range(3)):
    dot(i,j)
```



You might imagine that the optimal *integer* solution is just obtained by rounding. Note the following:

$(2, 2)$ is infeasible.

$(2, 1)$ is feasible and $f(2, 1) = 2 + 5 \cdot 1 = 7$

$(1, 2)$ is infeasible

$(1, 1)$ is feasible and $f(1, 1) = 1 + 5 \cdot 1 = 6$

But as it turns out, the optimal integer solution is the point $(0, 2)$ for which $f(0, 2) = 0 + 5 \cdot 2 = 10$.

Of course, this optimal integral solution is nowhere near the optimal non-integral solution. So in general, rounding is inadequate!

How to proceed? Well, in this case there are not very many integral feasible points, so to optimize, we can just check the value of f at all such points!

Consider a linear program in standard form for $\mathbf{x} \in \mathbb{R}^n$ with $\mathbf{x} \geq \mathbf{0}$, with inequality constraint $A\mathbf{x} \leq \mathbf{b}$ which seeks to `maximize` its objective function f.

Here is a systematic way that we might proceed:

Find an integer $M \geq 0$ with the property that

$$\mathbf{x} > M \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \implies \mathbf{x} \text{ is infeasible.}$$

There are $(M+1)^n$ points $\mathbf{x}$ with integer coordinates for which $\mathbf{0} \leq \mathbf{x} \leq M \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$.

For each of these points $\mathbf{x}$, we do the following: - if $\mathbf{x}$ is infeasible, discard - otherwise, record the pair $(\mathbf{x}, f(\mathbf{x}))$.

When we are finished, we just scan the list of recorded pairs and select that with the largest objective function value; this selection solves the problem.

The strategy just described is systematic, easy to describe, and works OK when $(M+1)^n$ isn't so large. But e.g. if $M = 3$ and $n = 20$, then already

$$(M+1)^n \approx 1.1 \times 10^{12},$$

which gives us a huge number of points to check!!!

4 A more efficient approach: "Branch & Bound"

We are going to describe an algorithm that implements a branch-and-bound strategy to approach the problem described above.

Let's fix some notation; after we formulate some generalities, we'll specialize our treatment to some examples.

4.1 Notation

We consider an `integer linear program`:

($\clubsuit$) `maximize` $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$

subject to:

- $\mathbf{x} \in \mathbb{Z}^n, \mathbf{x} \geq \mathbf{0}$
- $A\mathbf{x} \leq \mathbf{b}$ for some $A \in \mathbb{R}^{r \times n}$ and $\mathbf{b} \in \mathbb{R}^r$.

Recall that $\mathbb{Z} = \{0, \pm 1, \pm 2, \cdots\}$ is the set of *integers*, and $\mathbb{Z}^n$ is the just the set of vectors $\begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix}^T$ where $a_1, a_2, \dots, a_n \in \mathbb{Z}$.

We are going to suppose that we have some vector

$$\mathbf{M} = \begin{bmatrix} m_1 & m_2 & \cdots & m_n \end{bmatrix}^T \in \mathbb{Z}^n, \quad \mathbf{M} \geq \mathbf{0}$$

with the property that $\mathbf{x} > \mathbf{M} \implies \mathbf{x}$ is infeasible (i.e. $\mathbf{x} > \mathbf{M} \implies A\mathbf{x} > \mathbf{b}$).

In practice, it'll often be the case that $m_1 = m_2 = \cdots = m_n$ but that isn't a requirement for us.

Let's write

$$S = \{\mathbf{x} \in \mathbb{Z}^n \mid \mathbf{0} \leq \mathbf{x} \leq \mathbf{M}\}.$$

Note that the number of elements $|S|$ in the set S is given by the product

$$S = \prod_{i=1}^{n}(m_i + 1) = (m_1 + 1) \times (m_2 + 1) \times \cdots \times (m_n + 1).$$

And according to our assumption, S contains every feasible point $\mathbf{x}$ whose coordinates are integers. So a brute force approach to finding an optimal integral point $\mathbf{x}$ could be achieved just by testing each element of S.

Our goal is to systematically eliminate many of the points in S.

4.2 Algorithm overview

Keep the preceding notations. We sometimes refer to the entries x_i of $\mathbf{x}$ as "variables".

In the algorithm, we are going to keep track of a `search_queue` which is initially empty: `[]` and we are going to keep track of a `candidate_solution` which is initially `None`.

Let's focus on one entry of $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}^T \in \mathbb{Z}^n$, say the j-th entry for some $1 \leq j \leq n$ (we'll say more below about how we should select j). i.e. we focus on the variable x_j.

Now, x_j may take the values $0, 1, 2, \cdots, m_j$, so we consider the following subsets of S:

$$\begin{array}{ccc} S_0 & = & \{\mathbf{x} \in S \mid x_j = 0\} \\ S_1 & = & \{\mathbf{x} \in S \mid x_j = 1\} \\ \vdots & \vdots & \vdots \\ S_{m_j} & = & \{\mathbf{x} \in S \mid x_j = m_j\} \end{array}$$

Thus we have *partitioned* S as a *disjoint union* of certain subsets:

$$S = S_0 \cup S_1 \cup \cdots \cup S_{m_j}$$

For $0 \leq i \leq m_j$, consider the (relaxed) linear program $\mathcal{L}_i$ determined by `(c,A_ub,b_ub)` together with the equality constraint `xj=i`. Let's write `h_i` for the optimal value determined by solving this linear program, and `x_i` for a solution.

Now we loop `i` between `0` and `m_j`.

At each step of the loop, we check whether `x_i` is an integer solution.

If `x_i` is integral, computer the value `h_i` of the objective function at `x_i` with the value of the objective function at the current `candidate_solution`. If the value at `x_i` is higher, we replace the `candidate_solution` with `x_i`. Then we scan through the `search_queue`. For each item in the search queue, if `h_i` exceeds the value of the item, we remove it from the search queue.

If `x_i` is not integral and if `h_i` exceeds the value of the objective function at the current `candidate_solution`, we then add the data

```
{ constraints = [ (j,i) ],
  value = h_i
}
```

to the search queue and proceed with the loop.

When this loop on `i` is completed, we choose an element (e.g., the first element) of the `search_queue`, say it is

```
{ constraints j= [ (j0,i0) ],
  value = h_i0
}
```

As before, we consider a loop of `i` between `0` and `m_l`. Now we consider the linear program $\mathcal{L}_i$ determined by `(c,A_ub,b_ub)` toegether with the equality constraints `x_j0 = i0` and `x_l = i`; here we get optimal values `h_i` and solutions `x_i`.

At each step of this loop, we check whether `x_i` is an integer solution.

If `x_i` is integral, compare the value `h_i` of the objective function at `x_i` with the value of the objective function at the current `candidate_solution`. If the value at `x_i` is higher, we replace the `candidate_solution` with `x_i`. Then we scan through the `search_queue`. For each item in the search queue, if `h_i` exceeds the value of the item, we remove it from the search queue.

If `x_i` is not integral and if `h_i` exceeds the value of the objective function at the current `candidate_solution`, we then add the data

```
{ constraints = [ (l,i), (j0,i0) ],
  value = h_i0
}
```

to the search queue and proceed with the loop.

We can continue in this way; notice that the entries in the `search_queue` "remember" all the constraints that have been made "above it".

Eventually this process will result in an empty `search_queue` and at that point the `candidate_solution` is the actual solution.

As a guiding heuristic, at the start of the algorithm – and when we branch on items in the `search_queue` – we choose the variable on which we branch by finding the solution value which is non-integral but closest to a integer.

We use some code to simplify the process of solving the linear program with additional equality constraints for various variables.

[42]:
```
import numpy as np

def sbv(index,size):
    return np.array([1.0 if i == index else 0.0 for i in range(size)])

# description of linear program as dictionary has this format:
# lp = { "goal": "maximize",  # or "minimize"...
#        "obj": ...,          # remaining fields should be of the form np.
↪array(...)
#        "Aub": ...,
#        "bub": ...
#    }

# and we need to pass a list of equality constraints, each of the form
# {"index": i, "value": v}
# This dictionary represents the equality constraint "x_i = v"

def get_optimal(lp,specs = []):
    n = len(lp["obj"])

    Aeq = np.array([sbv(spec["index"],n) for spec in specs])
    beq = np.array([spec["value"] for spec in specs])

    #print(Aeq,beq)
    sgn = -1 if lp["goal"]=="maximize" else 1
    result = linprog(sgn*lp["obj"],
                     A_ub=lp["Aub"],
                     b_ub=lp["bub"],
                     A_eq = Aeq if not(specs==[]) else None,
                     b_eq = beq if not(specs==[]) else None)

    if result.success:
        return (sgn*result.fun,result.x)
    else:
        return "lin program failed"
```

4.3 Example

Consider again the integer linear program

$( ) $ maximize $f(x_0, x_1) = x_0 + 5x_1$; i.e. $\mathbf{c} \cdot \mathbf{x}$ where $\mathbf{c} = \begin{bmatrix} 1 & 5 \end{bmatrix}$.

such that $A\mathbf{x} = \begin{bmatrix} 1 & 10 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \leq \begin{bmatrix} 20 \\ 2 \end{bmatrix}$ and $\mathbf{x} \geq \mathbf{0}$ for $\mathbf{x} \in \mathbb{Z}^2$.

We notice that $(*) \quad \mathbf{x} > \begin{bmatrix} 2 & 2 \end{bmatrix}^T \implies \mathbf{x}$ is not feasible.

To begin, we first solve the linear program obtained from $(\diamondsuit)$ by considering $\mathbf{x} \in \mathbb{R}^2$.

Here is our dictionary representation of this linear program

[39]:
```
lp = { "goal": "maximize",
       "obj": np.array([1,5]),
       "Aub": np.array([[1,10],[1,0]]),
       "bub": np.array([20,2]),
       "bounds": 2*[(0,None)]
     }
```

And we can get a ("relaxed") solution using the above code, with `specs = []` – i.e. no equality constraints.

[44]:
```
val,sol = get_optimal(lp)

print(f"The optimal value is v = {val} and an optimal solution is {sol}")
```

```
The optimal value is v = 11.0 and an optimal solution is [2.  1.8]
```

Thus for this optimal solution, x_0 is already an integer, so we branch on x_1.

Now we branch on `x1`. Recall that – according to (∗) – we need only consider values of `x1` in `[0,1,2]`.

[46]:
```
[ (v,get_optimal(lp,[{"index":1, "value": v}])) for v in [0,1,2] ]
```

[46]:
```
[(0, (2.0, array([ 2., -0.]))),
 (1, (7.0, array([2., 1.]))),
 (2, (10.0, array([-0.,  2.])))]
```

Thus, we first add both

```
{ constraint: [ (1,0) ],
  value: 2.0
},
{ constraint: [ (1,1) ],
  value: 7.0
}
```

to `search_queue`. When we arrive `i=2`, we find an *integral* solution `x = [0,2]` with value 10.0. So we prune both of the two preceding items from the `search_queue` and insert `[0,2]` as the `candidate_solution`.

Now we notice that `search_queue == []` and so the `candidate_solution` is in fact the actual solution.

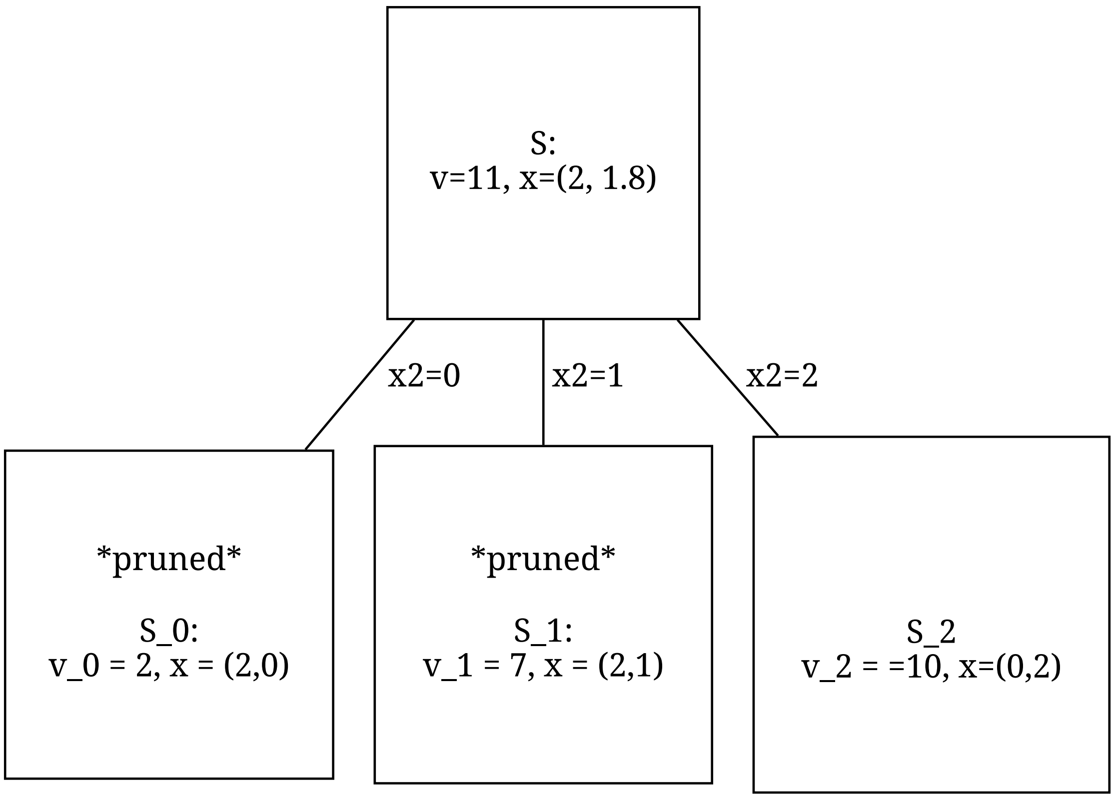
Recall that – in the notation used above – `fi` denotes the maximum value of the objective function on points having `x1 == i`. The preceding calculation shows that

One often presents this algorithm via a tree diagram, like the following:

[47]:
```
from graphviz import Graph

## https://www.graphviz.org/
## https://graphviz.readthedocs.io/en/stable/index.html
```

```
dot = Graph('bb1')

dot.node('S','S:\nv=11, x=(2, 1.8)',shape="square")

dot.node('S0','*pruned*\n\nS_0:\nv_0 = 2, x = (2,0)',shape="square")
dot.node('S1','*pruned*\n\nS_1:\nv_1 = 7, x = (2,1) ',shape="square")
dot.node('S2','\n\nS_2\nv_2 = =10, x=(0,2)',shape="square")


dot.edge('S','S0','x2=0')
dot.edge('S','S1','x2=1')
dot.edge('S','S2','x2=2')

dot
```

[47]:



Now let's consider a more elaborate example.

5 Example

$( ) $ `maximize` $f(\mathbf{x}) = \begin{bmatrix}10 & 7 & 4 & 3 & 1 & 0\end{bmatrix} \cdot \mathbf{x}$

subject to: $\mathbf{x} = \begin{bmatrix}x_1 & x_2 & x_3 & x_4 & x_5 & x_6\end{bmatrix}^T \in \mathbb{R}^5$, $\mathbf{x} \geq \mathbf{0}$,

$x_1, x_2, x_3, x_4, x_5 \in \{0, 1\}$

and $A\mathbf{x} \leq \mathbf{b}$

$$\text{where } A = \begin{bmatrix} 2 & 6 & 1 & 0 & 0 & 1 \\ 1 & 0 & 2 & -3 & 1 & -1 \\ 2 & -3 & 4 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & -1 & 0 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 7 \\ -1 \\ 3 \\ 3 \end{bmatrix}.$$

Notice that we *aren't* imposing any integral condition on x_6, but we require that $x_i \in \mathbb{Z}$ for $1 \leq i \leq 5$, and even more: these coordinates may only take the value 0 or 1.

The procedure described above can (with perhaps some minor adaptations) be applied to this problem, as we now describe. Note that – unlike the previous example – we will have to iterate our procedure.

We begin by formulating the linear program which replaces the integrality condition $x_1, x_2, x_3, x_4, x_5 \in \{0, 1\}$ with the condition $\begin{bmatrix}x_1 & x_2 & x_3 & x_4 & x_5\end{bmatrix}^T \leq \begin{bmatrix}1 & 1 & 1 & 1 & 1\end{bmatrix}^T$

[8]:
```
nlp = {"goal": "maximize",
       "obj": np.array([10,7,4,3,1,0]),
       "Aub": np.array([[2,6,1,0,0,1],
                        [1,0,2,-3,1,-1],
                        [2,-3,4,1,1,0],
                        [1,1,1,1,-1,0]]),
       "bub": np.array([7,-1,3,3]),
       "bounds": 5*[(0,1)] + [(0,None)]
      }
```

Solving this linear program yields the following:

[50]:
```
get_optimal(nlp)
```

[50]:
```
(24.71875, array([1.71875 , 0.59375 , 0.       , 1.015625, 0.328125, 0.      ]))
```

We are going to label this solution **(A)**

Of the non-integer coordinates of the solution `x` to **(A)**, the one closest to an integer is `x4 = 0.21`. (Remember that we don't impose an integrality condition on `x5`!!)

We now **Branch on (A)** with `x4`:

We label **(B)** the result of setting `x4==0` (and we add it to our `search_queue`):

[52]:
```
## B
```

```
resB =  get_optimal(nlp,[{"index": 4, "value": 0}])
resB
```

[52]:
```
(23.545454545454543,
 array([ 1.81818182,  0.45454545,  0.        ,  0.72727273, -0.        ,
         0.63636364]))
```

And we label **(C)** the result of setting `x4==1` (and we add it to our `search_queue`):

[53]:
```
## C

resC = get_optimal(nlp,[{"index": 4, "value": 1}])
resC
```

[53]:
```
(24.142857142857142,
 array([1.14285714, 0.78571429, 0.        , 2.07142857, 1.        ,
        0.        ]))
```

We must now branch off of both **(B)** and **(C)**.

We'll begin with **(C)** and branch on `x2`

(D) will be the result of branching from **(C)** with `x2==0` (and the results are added to our `search_queue`):

[54]:
```
## D

resD = get_optimal(nlp,[{"index": 4, "value": 1},
                        {"index": 2, "value": 0}])

resD
```

[54]:
```
(24.142857142857142,
 array([1.14285714, 0.78571429, 0.        , 2.07142857, 1.        ,
        0.        ]))
```

(E) will be the result of branching from **(C)** with `x2==1` :

[13]:
```
## E

resE = branch([{"index": 4, "value": 1},
               {"index": 2, "value": 1}],nlp)

resE
```

[13]:
```
'lin program failed'
```

This branching failed because there are no feasible points. So in fact (E) is not added to `search_queue`. Effectively, (E) is *pruned*.

We now branch on **(D)** with `x1`

F will be the result of branching on **(D)** with `x1==0`:

[55]:
```
## F

resF = get_optimal(nlp,[{"index": 4, "value": 1},
                        {"index": 2, "value": 0},
                        {"index": 1, "value": 0}])

resF
```

[55]:
```
(11.0, array([ 1., -0.,  0.,  0.,  1.,  3.]))
```

Note that **(F)** gives an integral solution. So now our `candidate_solution` is `[ 1., -0.,  0., -0.,  1.,  3.]` (replacing `None`), although the objective function value at this solution does not permit us to prune any entries in the `search_queue`.

Now, **(G)** will be the result of branching on **(D)** with `x1==1`:

[56]:
```
## G

resG = get_optimal(nlp,[{"index": 4, "value": 1},
                        {"index": 2, "value": 0},
                        {"index": 1, "value": 1}])

resG
```

[56]:
```
(20.5, array([0.5, 1. , 0. , 2.5, 1. , 0. ]))
```

We add this result to the `search_queue`.

We now return and **branch on (B)** with `x4`:

(H) will be the result of branching on **(B)** with `x3==0`:

[57]:
```
## H

resH = get_optimal(nlp,[{"index": 4, "value": 0},
                        {"index": 3, "value": 0}])

resH
```

[57]:
```
(18.142857142857146,
 array([ 1.71428571,  0.14285714,  0.        , -0.        ,  0.        ,
         2.71428571]))
```

this is again added to the `search_queue`.

(I) will be the result of branching on **(B)** with `x3==1`:

[58]:
```
## I
```

```python
resI = get_optimal(nlp,[{"index": 4, "value": 0},
                        {"index":3, "value": 1}])

resI
```

[58]: (21.8, array([1.6, 0.4, 0. , 1. , -0. , 0.]))

Now **branch on (I)** with `x1`:

(J) will be the result of branching from **(I)** with `x1=0`.

[59]:
```python
## J

resJ = get_optimal(nlp,[{"index": 4, "value": 0},
                        {"index":3, "value": 1},
                        {"index":1, "value": 0}])

resJ
```

[59]: (13.0, array([1., -0., 0., 1., 0., 0.]))

The solution **(J)** is an integer point, and the objective value 13 is larger than the objective value for the current `candidate_solution`, so we replace the `candidate_solution` by `[ 1., -0., -0., 1., -0.,  0.]`. (This effectively *prunes* (F)).

(K) will be the result of branching from **(I)** with `x1=1`:

[63]:
```python
## K

resK = get_optimal(nlp,[{"index": 4, "value": 0},
                        {"index": 3, "value": 1},
                        {"index": 1, "value": 1}])

resK
```

[63]: (15.0, array([0.5, 1. , 0. , 1. , -0. , 0.]))

Now **branch from (G)** with x_1:

(L) will be the result of branching from **(G)** with `x0=0`

[68]:
```python
## L

resL = get_optimal(nlp,[{"index": 4, "value": 1},
                        {"index": 2, "value": 0},
                        {"index": 1, "value": 1},
                        {"index": 0, "value": 0}])

resL
```

[68]: (17.0, array([-0., 1., -0., 3., 1., 0.]))

This is an integer solution. Since 17 exceeds 13, our new `candidate_solution` is `[-0.,  1., -0., 3.,  1.,  0.]`. Thus we effectively prune (J).

(M) is the result of branching from **(G)** with $x_0 = 1$

[65]:
```
## M

resM = get_optimal(nlp,[{"index": 4, "value": 1},
                        {"index": 2, "value": 0},
                        {"index": 1, "value": 1},
                        {"index": 0, "value": 1}])

resM
```

[65]:
```
'lin program failed'
```

The linear program **(M)** is infeasible.

Finally, **branch from (K)** with x_1

(N) is the result of branching from **(K)** with `x0=0`:

[66]:
```
## N



resN = get_optimal(nlp,[{"index": 4, "value": 0},
                        {"index":3, "value": 1},
                        {"index": 1, "value": 1},
                        {"index": 0, "value": 0}])

resN
```

[66]:
```
(14.0, array([-0.,  1.,  1.,  1., -0.,  0.]))
```

(N) gives an integer solution, but the objective function value of 14 does not exceed the objective function value of the current `candidate_solution`. So we prune (N).

(O) is the result of branching from **(K)** with `x0=1`:

[69]:
```
## O



resO = get_optimal(nlp,[{"index": 4, "value": 0},
                        {"index":3, "value": 1},
                        {"index": 1, "value": 1},
                        {"index": 0, "value": 1}])
resO
```

[69]:
```
'lin program failed'
```

This linear program is infeasible.

We are now **finished**; the integer solution from **(L)**, namely

[79]:
```
resL
```

[79]:
```
(17.0, array([-0.,  1., -0.,  3.,  1.,  0.]))
```

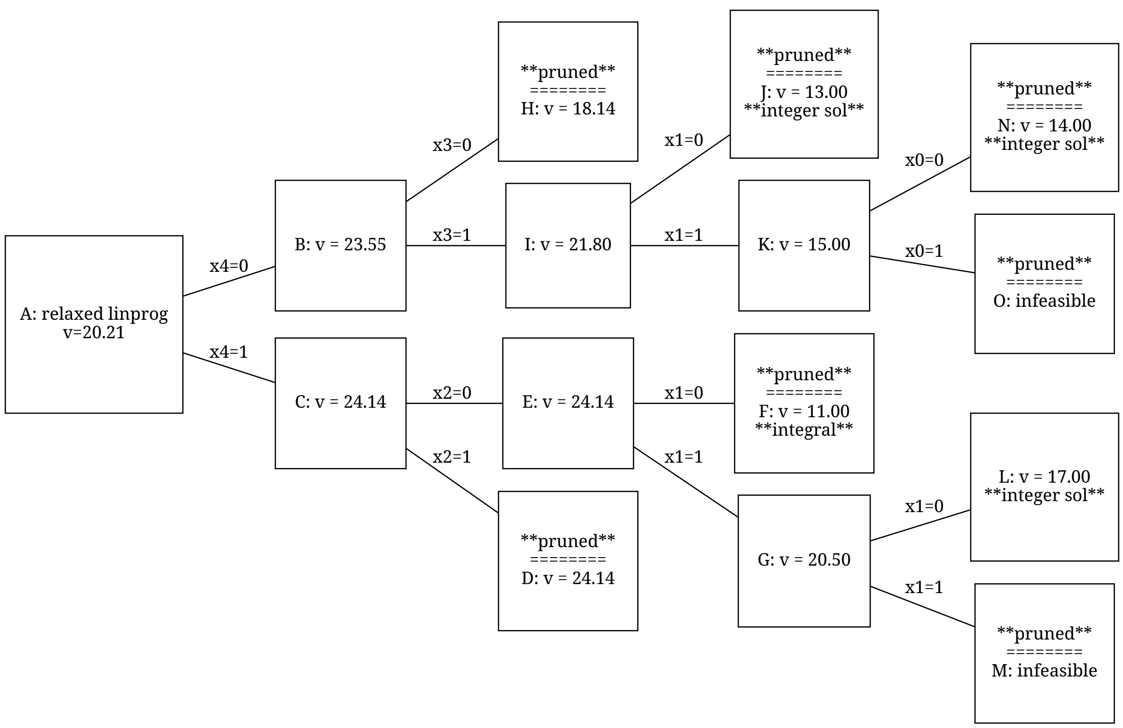
is the optimal integral solution.

We now construct (via `graphviz`) the *tree* corresponding to the branch-and-bound just carried out; see below.

[78]:
```
from graphviz import Graph

## https://www.graphviz.org/
## https://graphviz.readthedocs.io/en/stable/index.html

dot = Graph('bb1')
dot.attr(rankdir='LR')
dot.node('A',f"A: relaxed linprog\nv={resA['obj_value']:.2f}",shape="square")

dot.node('B',f"B: v = {resB[0]:.2f}",shape="square")
dot.node('C',f"C: v = {resC[0]:.2f}",shape="square")
dot.node('D',f"E: v = {resD[0]:.2f}",shape="square")
dot.node('E',f"**pruned**\n========\nD: v = {resD[0]:.2f}",shape="square")
dot.node('F',f"**pruned**\n========\nF: v = {resF[0]:.
 ↪2f}\n**integral**",shape="square")
dot.node('G',f"G: v = {resG[0]:.2f}",shape="square")
dot.node('H',f"**pruned**\n========\nH: v = {resH[0]:.2f}",shape="square")
dot.node('I',f"I: v = {resI[0]:.2f}",shape="square")
dot.node('J',f"**pruned**\n========\nJ: v = {resJ[0]:.2f}\n**integer␣
 ↪sol**",shape="square")
dot.node('K',f"K: v = {resK[0]:.2f}",shape="square")
dot.node('L',f"L: v = {resL[0]:.2f}\n**integer sol**",shape="square")
dot.node('M','**pruned**\n========\nM: infeasible',shape="square")
dot.node('N',f"**pruned**\n========\nN: v = {resN[0]:.2f}\n**integer␣
 ↪sol**",shape="square")
dot.node('O','**pruned**\n========\nO: infeasible',shape="square")


dot.edge('A','B','x4=0')
dot.edge('A','C','x4=1')

dot.edge('B','H','x3=0')
dot.edge('B','I','x3=1')

dot.edge('C','D','x2=0')
dot.edge('C','E','x2=1')
```

```
dot.edge('D','F','x1=0')
dot.edge('D','G','x1=1')

dot.edge('G','L','x1=0')
dot.edge('G','M','x1=1')

dot.edge('I','J','x1=0')
dot.edge('I','K','x1=1')


dot.edge('K','N','x0=0')
dot.edge('K','O','x0=1')




dot
```

[78]:



6 Postcript

It turns out that solving integer programming problems is *hard*. In fact, in computer science integer programming problems are in a class of problems called **NP Hard** problems – see the discussion here..

The algorithm we describe above is a type of `branch and bound algorithm`, which is a common approach. While our description gives pretty good evidence that this approach is effective, we haven't said anything e.g. about the `run time` of our algorithm, etc.