# Branch & Bound and Knapsack Lab

**Objectives**

- Preform the branch and bound algorithm
- Apply branch and bound to the knapsack problem
- Understand the geometry of the branch and bound algorithm

**Brief description:** In this lab, we will try solving an example of a knapsack problem with the branch-and-bound algorithm. We will also see how adding a cutting plane helps in reducing the computation time and effort of the algorithm. Lastly, we will explore the geometry of the branch and bound algorithm.

In [83]:
```
# imports -- don't forget to run this cell
import pandas as pd
import gilp
from gilp.visualize import feasible_integer_pts
from ortools.linear_solver import pywraplp as OR
```

## Part 1: Branch and Bound Algorithm

Recall that the branch and bound algorithm (in addition to the simplex method) allows us to solve integer programs. Before applying the branch and bound algorithm to the knapsack problem, we will begin by reviewing some core ideas. Furthermore, we will identify a helpful property that will make branch and bound terminate quicker later in the lab!

```
**Q1:** What are the different ways a node can be fathomed during the branch
and bound algorithm? Describe each.

**A:** (**For some reason the graphics won't show on the pdf, even though
they worked on the lab**) First, we can fathom a node if we already know a
solution that will be better than or equal to both the branches of the
current node. We can also fathom a node if we know that it wont give any
feasibly solutions in its branches.
```

**Q2:** Suppose you have a maximization integer program and you solve its linear program relaxation. What does the LP-relaxation optimal value tell you about the IP optimal value? What if it is a minimization problem?

**A:** The LP relaxation optimal value tells you bounds for the IP optimal value. For example, a maximization LP gives an upper bound for the IP optimal value, and the minimization LP gives a lower bound for the IP optimal value.

**Q3:** Assume you have a maximization integer program with all integral coefficents in the objective function. Now, suppose you are running the branch and bound algorithm and come across a node with an optimal value of 44.5. The current incumbent is 44. Can you fathom this node? Why or why not?

**A:** You can fathom this node because we already know that the current incumbent is 44, and there can be no better outcome from 44.5 than 44.

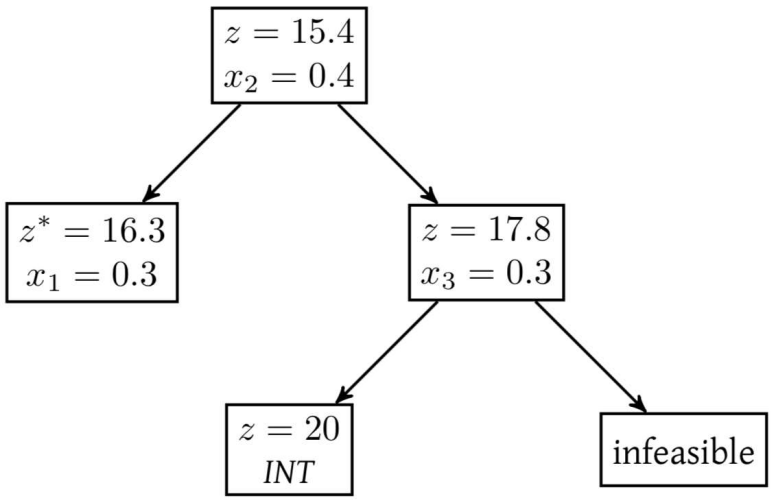
**Q4:** If the optimal solution to the LP relaxation of the original program is integer, then you have found an optimal solution to your integer program. Explain why this is true.

**A:** That is because when you add the constraint that every variable must be integral, that is adding a constraint to the original LP, which means that the objective value is always going to be less than or equal to the current optimal value to the LP. Thus, when the solution to the original LP is already integer, adding the constraint that the objective value must be integer changes nothing.

**Q5:** If the LP is infeasible, then the IP is infeasible. Explain why this is true.

**A:** Once again, we are adding an extra constraint when going from the LP to the IP. Thus, if the LP is infeasible, adding a constraint will change nothing because all the other existing constraints that make the original LP are still present.

The next questions ask about the following branch and bound tree. If the solution was not integral, the fractional $x_i$ that was used to branch is given. If the solution was integral, it is denoted *INT*. In the current iteration of branch and bound, you are looking at the node with the *****.




**Q6:** Can you determine if the integer program this branch and bound tree is for is a minimization or maximixation problem? If so, which is it?

**A:** This is for a minimzation problem because the objective functions are increasing as you go down the tree (which won't happen in a mazimization problem).

Hint: For **Q7-8**, you can assume integral coefficents in the objective function.

**Q7:** Is the current node (marked $z^*$) fathomed? Why or why not? If not, what additional constraints should be imposed for each of the next two nodes?

**A:** The current node is not fathomed because there could be an objective value that results from its two branches that results in a better solution than 20. The additional constraints that should be imposed for each of the next two nodes are that X1 be rounded down and rounded up for each of the nodes in order to insure that the variable is an integer value

**Q8:** Consider the nodes under the current node (where $z = 16.3$). What do you know about the optimal value of these nodes? Why?

**A:** The optimal value must be greater than 16.3 because you are adding a constraint to this current minimization problem, which can only make the value increase.

# Part 2: The Knapsack Problem

In this lab, you will solve an integer program by branch and bound. The integer program to be solved will be a knapsack problem.

**Knapsack Problem:** We are given a collection of $n$ items, where each item $i = 1, \dots, n$ has a weight $w_i$ and a value $v_i$. In addition, there is a given capacity $W$, and the aim is to select a maximum value subset of items that has a total weight at most $W$. Note that each item can be brought at most once.

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \leq W \\ & 0 \leq x_i \leq 1, \text{integer}, i = 1, \dots, n \end{aligned}$$

Consider the following data which we import from a CSV file:

In [51]:
```
data = pd.read_csv('knapsack_data_1.csv', index_col=0)
data
```

Out[51]:

| item | value | weight |
|---|---|---|
| **1** | 50 | 10 |
| **2** | 30 | 12 |
| **3** | 24 | 10 |
| **4** | 14 | 7 |
| **5** | 12 | 6 |
| **6** | 10 | 7 |
| **7** | 40 | 30 |

and $W = 18$.

**Q9:** Are there any items we can remove from our input to simplify this problem? Why? If so, replace `index` with the item number that can be removed in the code below. Hint: how many of each item could we possibly take?

**A:** We can remove item 7 because, by itself, it exceeds the constraint that the total weight must less than or equal to 18.

In [52]:
```
# TODO: replace index
data = data.drop(7)
```

**Q10:** If we remove item 7 from the knapsack, it does not change the optimal solution to the integer program. Explain why.

**A:** Because item 7 cannot be a part of any feasible solution since its own weight alone exceeds the maximum weight constraint.

**Q11:** Consider removing items $i$ such that $w_i > W$ from a knapsack input. How does the LP relaxation's optimal value change?

**A:** It does not change at all because they are never a part of any feasible solution

In **Q10-11**, you should have found that removing these items removes feasible solutions from the linear program but does not change the integer program. This is desirable as the gap between the optimal IP and LP values can become smaller. By adding this step, branch and bound may

terminate sooner.

Recall that a branch and bound node can be fathomed if its bound is no better than the value of the best feasible integer solution found thus far. Hence, it helps to have a good feasible integer solution as quickly as possible (so that we stop needless work). To do this, we can first try to construct a good feasible integer solution by a reasonable heuristic algorithm before starting to run the branch and bound procedure.

In designing a heuristic for the knapsack problem, it is helpful to think about the value per unit weight for each item. We compute this value in the table below.

In [53]:
```
data['value per unit weight'] = (data['value'] / data['weight']).round(2)
data
```

Out[53]:

| item | value | weight | value per unit weight |
|---|---|---|---|
| **1** | 50 | 10 | 5.00 |
| **2** | 30 | 12 | 2.50 |
| **3** | 24 | 10 | 2.40 |
| **4** | 14 | 7 | 2.00 |
| **5** | 12 | 6 | 2.00 |
| **6** | 10 | 7 | 1.43 |

**Q12:** Design a reasonable heuristic for the knapsack problem. Note a heuristic aims to find a decent solution to the problem (but is not necessarily optimal).

**A:** You first choose the item of the highest value that is lighter than the weight you are allowed to carry, and then try to take the next highest value item that you can still fit into the bag and satisfy the weight constraint. You do this until you can't put anymore items in.

**Q13:** Run your heuristic on the data above to compute a good feasible integer solution. Your heuristic should generate a feasible solution with a value of 64 or better. If it does not, try a different heuristic (or talk to your TA!)

**A:** This heuristic generated a result of 64, choosing items 1 and 4.

We will now use the branch and bound algorithm to solve this knapsack problem! First, let us define a mathematical model for the linear relaxation of the knapsack problem.

**Q14:** Complete the model below.

In [54]:
```
def Knapsack(table, capacity, integer = False):
    """Model for solving the Knapsack problem.

    Args:
        table (pd.DataFrame): A table indexd by items with a column for value
        capcity (int): An integer-capacity for the knapsack
        integer (bool): True if the variables should be integer. False otherw
    """
    ITEMS = list(table.index)        # set of items
    v = table.to_dict()['value']     # value for each item
    w = table.to_dict()['weight']    # weight for each item
    W = capacity                     # capacity of the knapsack

    # define model
    m = OR.Solver('knapsack', OR.Solver.CBC_MIXED_INTEGER_PROGRAMMING)

    # decision variables
    x = {}
    for i in ITEMS:
        if integer:
            x[i] = m.IntVar(0, 1, 'x_%d' % (i))
        else:
            x[i] = m.NumVar(0, 1, 'x_%d' % (i))

    # define objective function here
    m.Maximize(sum(v[i]*x[i] for i in ITEMS))

    # TODO: Add a constraint that enforces that weight must not exceed capaci
    # recall that we add constraints to the model using m.Add()
    m.Add(sum(w[i]*x[i] for i in ITEMS) <= W)

    return (m, x)  # return the model and the decision variables
```

In [55]:
```
# You do not need to do anything with this cell but make sure you run it!
def solve(m):
    """Used to solve a model m."""
    m.Solve()

    print('Objective =', m.Objective().Value())
    print('iterations :', m.iterations())
    print('branch-and-bound nodes :',m.nodes())

    return ({var.name() : var.solution_value() for var in m.variables()})
```

We can now create a linear relaxation of our knapsack problem. Now, `m` represents our model and `x` represents our decision variables.

In [56]:
```
m, x = Knapsack(data, 18)
```

We can use the next line to solve the model and output the solution

In [57]:
```
solve(m)
```

```
Objective = 70.0
iterations : 0
branch-and-bound nodes : 0
```

Out[57]:
```
{'x_1': 1.0,
 'x_2': 0.6666666666666667,
 'x_3': 0.0,
 'x_4': 0.0,
 'x_5': 0.0,
 'x_6': 0.0}
```

**Q15:** How does this optimal value compare to the value you found using the heuristic integer solution?

**A:** The optimal value is greater than the value found by the hearistic.

**Q16:** Should this node be fathomed? If not, what variable should be branched on and what additional constraints should be imposed for each of the next two nodes?

**A:** This node should not be fathomed. X2 should be branched on and the additional constraints should be that X2 will be an integer by either being 0 or 1.

After constructing the linear relaxation model using `Knapsack(data1, 18)` we can add additional constraints. For example, we can add the constraint $x_2 \leq 0$ and solve it as follows:

In [58]:
```
m, x = Knapsack(data, 18)
m.Add(x[2] <= 0)
solve(m)
```

```
Objective = 69.2
iterations : 0
branch-and-bound nodes : 0
```

Out[58]:
```
{'x_1': 1.0, 'x_2': 0.0, 'x_3': 0.8, 'x_4': 0.0, 'x_5': 0.0, 'x_6': 0.0}
```

**NOTE:** The line `m, x = Knapsack(data1, 18)` resets the model `m` to the LP relaxation. All constraints from branching have to be added each time.

**Q17:** Use the following cell to compute the optimal value for the other node you found in **Q16**.

In [59]:
```
m, x = Knapsack(data, 18)
m.Add(x[2] >= 1)
solve(m)
```

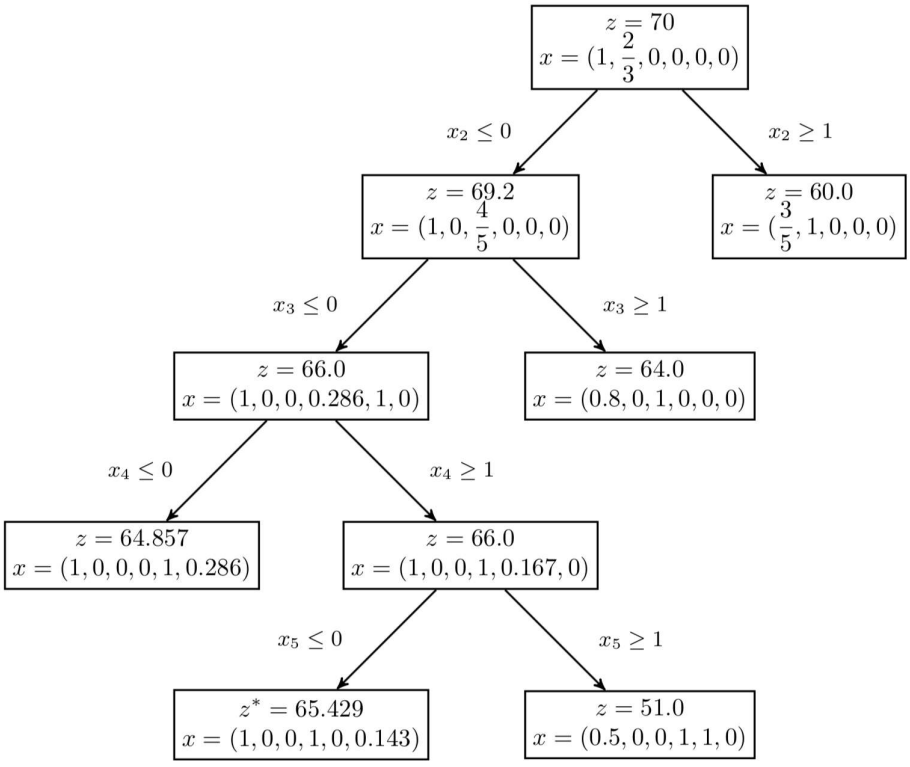
```
Objective = 60.0
iterations : 0
branch-and-bound nodes : 0
```

Out[59]:
```
{'x_1': 0.6000000000000001,
 'x_2': 1.0,
 'x_3': 0.0,
 'x_4': 0.0,
 'x_5': 0.0,
 'x_6': 0.0}
```

**Q18:** What was the optimal value? Can this node be fathomed? Why? (Hint: In **Q13**, you found a feasible integer solution with value 64.)

**A:** The optimal value is 60, which means that this can be fathomed because any feasibly solution to the Knapsack problem will have a value less than or equal to 60 which can be ignored since the heuristic found that there can be a feasible solution of 64.

If we continue running the branch and bound algorithm, we will eventually reach the branch and bound tree below where the $z^*$ indictes the current node we are looking at.




**Q19:** The node with $z = 64.857$ was fathomed. Why are we allowed to fathom this node? (Hint: think back to **Q3**)

**A:** We are allowed to fathom this node because the highest feasible solution that can come out of this is less than 64.857 and must be integer, which would be 64. Thus, this can be fathomed since we know that 64 is already feasible.

**Q20:** Finish running branch and bound to find the optimal integer solution. Use a separate cell for each node you solve and indicate if the node was fathomed with a comment. (Hint: Don't forget to include the constraints further up in the branch and bound tree.)

In [60]:
```
# Template
m, x = Knapsack(data, 18)
# Add constraints here
m.Add(x[2] <= 0)
m.Add(x[3] <=0 )
m.Add(x[4] >= 1)
m.Add(x[5] <= 0)
m.Add(x[6] <= 0)

solve(m)
# fathomed? Yes, because 64 was already found to be feasible.
```

```
Objective = 64.0
iterations : 0
branch-and-bound nodes : 0
```

Out[60]:
```
{'x_1': 1.0, 'x_2': 0.0, 'x_3': 0.0, 'x_4': 1.0, 'x_5': 0.0, 'x_6': 0.0}
```

In [61]:
```
# Template
m, x = Knapsack(data, 18)
# Add constraints here
m.Add(x[2] <= 0)
m.Add(x[3] <=0 )
m.Add(x[4] >= 1)
m.Add(x[5] <= 0)
m.Add(x[6] >= 1)

solve(m)
# fathomed? Yes, because 44 is less than 64, which was already found to be fe
```

```
Objective = 44.0
iterations : 0
branch-and-bound nodes : 0
```

Out[61]:
```
{'x_1': 0.4, 'x_2': 0.0, 'x_3': 0.0, 'x_4': 1.0, 'x_5': 0.0, 'x_6': 1.0}
```

In [ ]:
```

```

**A:**

**Q21:** How many nodes did you have to explore while running the branch and bound algorithm?

**A:** two nodes

In the next section, we will think about additional constraints we can add to make running branch and bound quicker.

# Part 3: Cutting Planes

In general, a cutting plane is an additional constraint we can add to an integer program's linear relaxation that removes feasible linear solutions but does not remove any integer feasible solutions. This is very useful when solving integer programs! Recall many of the problems we have learned in class have something we call the "integrality property". This is useful because it allows us to ignore the integrality constraint since we are garunteed to reach an integral solution. By cleverly adding cutting planes, we strive to remove feasible linear solutions (without removing any integer feasible solutions) such that the optimal solution to the linear relaxation is integral!

Conisder an integer program whose linear program releaxation is

$$\begin{aligned} \max \quad & 2x_1 + x_2 \\ \text{s.t.} \quad & x_1 + x_2 \leq 3 \\ & 2x_1 \leq 5 \\ & -x_1 + 2x_2 \leq 2 \\ & x_1, x_2 \geq 0 \end{aligned}$$

We can define this linear program and then visualize its feasible region. The integer points have been highlighted.

In [88]:
```
lp = gilp.LP([[1,1],[2,0],[-1,2]],
             [3,5,2],
             [2,1])
fig = gilp.lp_visual(lp)
fig.set_axis_limits([3.5,2])
fig.add_trace(feasible_integer_pts(lp, fig))
fig
```

**Q22:** List every feasible solution to the integer program.

**A:** (0,0), (0,1), (1,0), (1,1), (2,0), (2,1)

**Q23:** Is the constraint $x_2 \leq 1$ a cutting plane? Why? (Hint: Would any feasible integer points become infeasible? What about feasible linear points?)

**A:** Yes, this is a cutting plant because it doesn't make any current feasible integer points infeasible.

Let's add this cutting plane to the LP relaxation!

In [63]:
```
lp = gilp.LP([[1,1],[2,0],[-1,2],[0,1]],
             [3,5,2,1],
             [2,1])
fig = gilp.lp_visual(lp)
fig.set_axis_limits([3.5,2])
fig.add_trace(feasible_integer_pts(lp, fig))
fig
```

**Q24:** Is the constraint $x_1 \leq 3$ a cutting plane? Why?

**A:** No, because X1 is already constrained to be less than 2.5

**Q25:** Can you provide another cutting plane? If so, what is it?

**A:** Yes, X2 <= 2

Let's look at the feasible region after adding the cutting plane from **Q23** and one of the possible answers from **Q25**. Notice the optimal solution to the LP relaxation is now integral!

In [64]:
```
lp = gilp.LP([[1,1],[2,0],[-1,2],[0,1],[1,0]],
             [3,5,2,1,2],
             [2,1])
fig = gilp.lp_visual(lp)
fig.set_axis_limits([3.5,2])
fig.add_trace(feasible_integer_pts(lp, fig))
fig
```

Let's try applying what we know about cutting planes to the knapsack problem! Again, recall our input was $W = 18$ and:

In [65]:
```
data
```

Out[65]:

| item | value | weight | value per unit weight |
|---|---|---|---|
| **1** | 50 | 10 | 5.00 |
| **2** | 30 | 12 | 2.50 |
| **3** | 24 | 10 | 2.40 |
| **4** | 14 | 7 | 2.00 |
| **5** | 12 | 6 | 2.00 |
| **6** | 10 | 7 | 1.43 |

**Q26:** Look at items 1, 2, and 3. How many of these items can we take simultaneously? Can you write a new constraint to capture this? If so, please provide it.

**A:** You can only take one simultaneously. X1 + X2 + X3 <= 1

**Q27:** Is the constraint you found in **Q26** a cutting plane? If so, provide a feasible solution to the linear program relaxation that is no longer feasible (i.e. a point the constraint *cuts off*).

**A:** Yes, this is a cutting plane because it cuts off any chance of taking part of an item. For example, X1 = 1 X2 = 2/3 X3 = 0 was feasible beforehand.

**Q28:** Provide another cutting plane involving items 4,5 and 6 for this integer program. Explain how you derived it.

**A:** X4 + X5 + X6 <= 2. This is because, in any way you pick the items, you can take at most 2 items simultaneously.

**Q29:** Add the cutting planes from **Q26** and **Q28** to the model and solve it. You should get a solution in which we take items 1 and 4 and $\frac{1}{6}$ of item 5 with an objective value of 66.

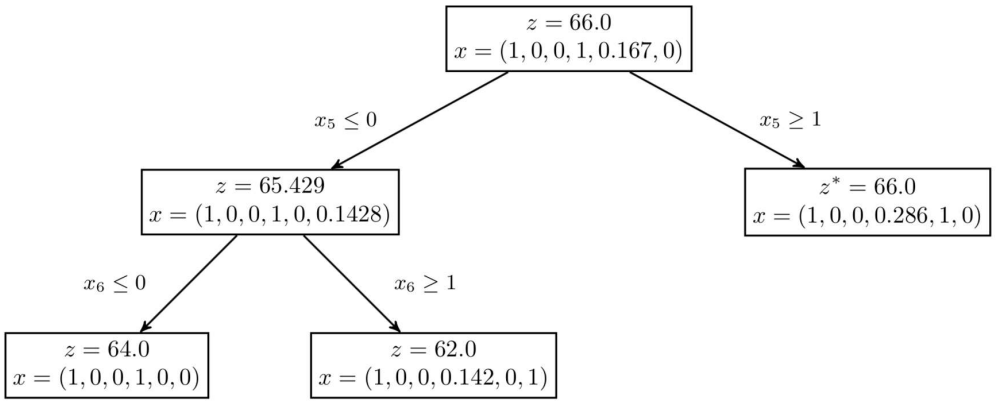
In [66]:
```
m, x = Knapsack(data, 18)
m.Add(x[1] + x[2] + x[3] <= 1)
m.Add(x[4] + x[5] + x[6] <= 2)

solve(m)
```

```
Objective = 66.0
iterations : 0
branch-and-bound nodes : 0
```

Out[66]:
```
{'x_1': 1.0,
 'x_2': 0.0,
 'x_3': 0.0,
 'x_4': 1.0,
 'x_5': 0.16666666666666696,
 'x_6': 0.0}
```

Let's take a moment to pause and reflect on what we are doing. Recall from **Q9-11** that we dropped item 7 becuase its weight was greater than the capcity of the knapsack. Essentially we added the constraint $x_7 \leq 0$. This constraint was a cutting plane! It eliminated some linear feasible solutions but no integer ones. By adding these two new cutting planes, we can get branch and bound to terminate earlier yet again! So far, we have generated cutting planes by inspection. However, there are more algorithmic ways to identify them (which we will ignore for now).

If we continue running the branch and bound algorithm, we will eventually reach the branch and bound tree below where the $z^*$ indictes the current node we are looking at.




**NOTE:** Do not forget about the feasible integer solution our heuristic gave us with value 64.

**Q30** Finish running branch and bound to find the optimal integer solution. Use a separate cell for each node you solve and indicate if the node was fathomed with a comment. Hint: Don't forget the cutting plane constraints should be included in every node of the branch and bound tree.

In [67]:
```
# Template
m, x = Knapsack(data, 18)
m.Add(x[5] >= 1)
m.Add(x[4] >= 1)

solve(m)
# fathomed? Yes, because 51 is less than 64
```

```
Objective = 51.0
iterations : 0
branch-and-bound nodes : 0
```

Out[67]:
```
{'x_1': 0.5, 'x_2': 0.0, 'x_3': 0.0, 'x_4': 1.0, 'x_5': 1.0, 'x_6': 0.0}
```

In [68]:
```
# Template
m, x = Knapsack(data, 18)
m.Add(x[5] >= 1)
m.Add(x[4] <= 0)

solve(m)
# fathomed? No
```

```
Objective = 67.0
iterations : 0
branch-and-bound nodes : 0
```

Out[68]:
```
{'x_1': 1.0,
 'x_2': 0.16666666666666674,
 'x_3': 0.0,
 'x_4': 0.0,
 'x_5': 1.0,
 'x_6': 0.0}
```

In [69]:
```
# Template
m, x = Knapsack(data, 18)
m.Add(x[5] >= 1)
m.Add(x[4] <= 0)
m.Add(x[2] >= 1)

solve(m)
# fathomed? Yes
```

```
Objective = 42.0
iterations : 0
branch-and-bound nodes : 0
```

Out[69]:
```
{'x_1': 0.0, 'x_2': 1.0, 'x_3': 0.0, 'x_4': 0.0, 'x_5': 1.0, 'x_6': 0.0}
```

In [70]:
```
# Template
m, x = Knapsack(data, 18)
m.Add(x[5] >= 1)
m.Add(x[4] <= 0)
m.Add(x[2] <= 0)

solve(m)
# fathomed? No
```

```
Objective = 66.80000000000001
iterations : 0
branch-and-bound nodes : 0
```

Out[70]:
```
{'x_1': 1.0,
 'x_2': 0.0,
 'x_3': 0.20000000000000018,
 'x_4': 0.0,
 'x_5': 1.0,
 'x_6': 0.0}
```

In [71]:
```
# Template
m, x = Knapsack(data, 18)
m.Add(x[5] >= 1)
m.Add(x[4] <= 0)
m.Add(x[2] <= 0)
m.Add(x[3] >= 1)

solve(m)
# fathomed? yes
```

```
Objective = 46.0
iterations : 0
branch-and-bound nodes : 0
```

Out[71]:
```
{'x_1': 0.2, 'x_2': 0.0, 'x_3': 1.0, 'x_4': 0.0, 'x_5': 1.0, 'x_6': 0.0}
```

In [72]:
```
# Template
m, x = Knapsack(data, 18)
m.Add(x[5] >= 1)
m.Add(x[4] <= 0)
m.Add(x[2] <= 0)
m.Add(x[3] <= 0)

solve(m)
# fathomed? No
```

```
Objective = 64.85714285714286
iterations : 0
branch-and-bound nodes : 0
```

Out[72]:
```
{'x_1': 1.0,
 'x_2': 0.0,
 'x_3': 0.0,
 'x_4': 0.0,
 'x_5': 1.0,
 'x_6': 0.28571428571428603}
```

In [73]:
```
# Template
m, x = Knapsack(data, 18)
m.Add(x[5] >= 1)
m.Add(x[4] <= 0)
m.Add(x[2] <= 0)
m.Add(x[3] <= 0)
m.Add(x[6] <= 0)

solve(m)
# fathomed? Yes
```

```
Objective = 62.0
iterations : 0
branch-and-bound nodes : 0
```

Out[73]:
```
{'x_1': 1.0, 'x_2': 0.0, 'x_3': 0.0, 'x_4': 0.0, 'x_5': 1.0, 'x_6': 0.0}
```

In [74]:
```
# Template
m, x = Knapsack(data, 18)
m.Add(x[5] >= 1)
m.Add(x[4] <= 0)
m.Add(x[2] <= 0)
m.Add(x[3] <= 0)
m.Add(x[6] >= 1)

solve(m)
# fathomed? Yes
```

```
Objective = 47.0
iterations : 0
branch-and-bound nodes : 0
```

Out[74]:
```
{'x_1': 0.5, 'x_2': 0.0, 'x_3': 0.0, 'x_4': 0.0, 'x_5': 1.0, 'x_6': 1.0}
```

**A:**

**Q31:** Did you find the same optimal solution? How many nodes did you explore? How did this compare to the number you explored previously?

**A:** Yes, the optimal solution (1, 0, 0, 1, 0, 0) with a value of 64. I had to explore 8 nodes, which is greater than the number of nodes before (two).

# Part 4: Geometry of Branch and Bound

Previously, we used the `gilp` package to viusualize the simplex algorithm but it also has the functionality to visualize branch and bound. We will give a quick overview of the tool. Similar to `lp_visual` and `simplex_visual`, the function `bnb_visual` takes an `LP` and returns a visualization. It is assumed that every decision variable is constrained to be integer. Unlike previous visualizations, `bnb_visual` returns a series of figures for each node of the branch and bound tree. Let's look at a small 2D example:

$$\begin{aligned} \max \quad & 5x_1 + 8x_2 \\ \text{s.t.} \quad & x_1 + x_2 \leq 6 \\ & 5x_1 + 9x_2 \leq 45 \\ & x_1, x_2 \geq 0, \quad \text{integral} \end{aligned}$$

In [75]:
```
nodes = gilp.bnb_visual(gilp.examples.STANDARD_2D_IP)
```

In [76]:
```
nodes[0].show()
```

nodes[2].show()

Run the cells above to generate a figure for each node and view the first node. At first, you will see the LP relaxation on the left and the root of the branch and bound tree on the right. The simplex path and isoprofit slider are also present.

**Q32:** Recall the root of a branch and bound tree is the unaltered LP relaxation. What is the optimal solution? (Hint: Use the objective slider and hover over extreme points).

**A:** The optimal solution is (2.25, 3.75) with an objective value of 41.25

**Q33:** Assume that we always choose the variable with the minimum index to branch on if there are multiple options. Write down (in full) each of the LPs we get after branching off the root node.

**A:** (1) Max 5X1 + 8X2 X1 + X2 <= 6 5X1 + 9X2 <= 45 solution: (2.25, 3.75) value: 41.25 (2) Max 5X1 + 8X2 X1 + X2 <= 6 5X1 + 9X2 <= 45 X1 <= 2 solution: (2, 3.889) value: 41.111 (3) Max 5X1 + 8X2 X1 + X2 <= 6 5X1 + 9X2 <= 45 X1 <= 2 X2 <= 3 solution: (2, 3) value: 34

**Q34:** Draw the feasible region to each of the LPs from **Q33** on the same picture.

**A:** Completed on paper

Run the following cell to see if the picture you drew in **Q34** was correct.

In [77]:
```
nodes[1].show()
```

The outline of the original LP relaxation is still shown on the left. Now that we have eliminated some of the fractional feasible solutions, we now have 2 feasible regions to consider. The darker one is the feasible region associated with the current node which is also shaded darker in the branch and bound tree. The unexplored nodes in the branch and bound tree are not shaded in.

**Q35:** Which feasible solutions to the LP relaxation are removed by this branch?

**A:** The feasible solutions that are removed are (3,3), (3,2), (3,1), (3,0), (4,2), (4,1), (4,0), (5,1), (5,0), (6,0)

**Q36:** At the current (dark) node, what constraints will we add? How many feasible regions will the original LP relaxation be broken into?

**A:** The next constraints that we will add are X2 <= 3 and X2 >= 4. After this, there are three feasible regions from the original input.

In [78]:
```
nodes[2].show()
```

**Q37:** What is the optimal solution at the current (dark) node? Do we have to further explore this branch? Explain.

**A:** The optimal solution is (2,3) with an objective value of 34. We do not need to explore anymore nodes because each of them can be fathomed since they will be less than or equal to 34.

**Q38:** Recall shaded nodes have been explored and the node shaded darker (and feasible region shaded darker) correspond to the current node and its feasible region. Nodes not shaded have not been explored. How many nodes have not yet been explored?

**A:** There are two nodes that have not yet been explored.

**Q39:** How many nodes have a degree of one in the branch and bound tree? (That is, they are only connected to one edge). These nodes are called leaf nodes. What is the relationship between the leaf nodes and the remaining feasible region?

**A:** There are three leaf nodes in the graphic above. The relationship between the leaf nodes and the remaining feasible region is that the remaining feasibly region at a certin point will always be a leaf node.

In [79]:

```
# Show the next two iterations of the branch and bound algorithm
nodes[3].show()
nodes[4].show()
```

**Q40:** At the current (dark) node, we added the constraint $x_1 \leq 1$. Why were the fractional solutions $1 < x_1 < 2$ not eliminated for $x_2 <= 3$?

**A:** Because the optimal solution and objective function value were already integers.

In [80]:

```
# Show the next three iterations of the branch and bound algorithm
nodes[5].show()
nodes[6].show()
nodes[7].show()
```

**Q41:** What constraints are enforced at the current (dark) node? Why are there no feasible solutions at this node?

**A:** The constraints are 2 <= X1 <= 2 and X2 >= 4. If you look at the graph, there are no feasible solutions that satisfy these constraints, which is why there are no feasible solutions at this node.

In [81]:
```
nodes[8].show()
```

**Q42:** Are we done? If so, what nodes are fathomed and what is the optimal solution? Explain.

**A:** Yes, we are done. The nodes that are fathomed are the ones with the corresponding objective values (39, 34, 40, 37). These are fathomed because their branches will all result in a value less than the optimal solution (0, 5) with an objective value of 40.

Let's look at branch and bound visualization for an integer program with 3 decision variables!

In [82]:
```
nodes = gilp.bnb_visual(gilp.examples.VARIED_BRANCHING_3D_IP)
```

```
C:\Users\jonas\.conda\envs\ENGRI_1101\lib\site-packages\gilp\simplex.py:435: LinAlgWarning:

Ill-conditioned matrix (rcond=4.40213e-36): result may not be accurate.


---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
<ipython-input-82-cec2dbb6f3c1> in <module>
----> 1 nodes = gilp.bnb_visual(gilp.examples.VARIED_BRANCHING_3D_IP)

~\.conda\envs\ENGRI_1101\lib\site-packages\gilp\visualize.py in bnb_visual(lp, manual, feas_tol, int_feas_tol)
    793                     trace = labeled_feasible_region(lp=feas_reg,
    794                                                     basic_sol=False,
--> 795                                                     show_basis=False)
    796                     fig.add_traces(trace)
    797             except Infeasible:

~\.conda\envs\ENGRI_1101\lib\site-packages\gilp\visualize.py in labeled_feasible_region(lp, theme, basic_sol, show_basis, vertices)
    283     region = feasible_region(lp=lp,
    284                              theme=theme,
--> 285                              vertices=vertices)
    286     bfs = bfs_plot(lp=lp,
    287                    basic_sol=basic_sol,

~\.conda\envs\ENGRI_1101\lib\site-packages\gilp\visualize.py in feasible_region(lp, theme, vertices)
    219     n,m,A,b,c = lp.get_coefficients(equality=False)
    220     try:
--> 221         simplex(LP(A,b,np.ones((n,1))))
    222     except UnboundedLinearProgram:
    223         raise InfiniteFeasibleRegion('Can not visualize.')

~\.conda\envs\ENGRI_1101\lib\site-packages\gilp\simplex.py in simplex(lp, pivot_rule, initial_solution, iteration_limit, feas_tol)
    606                                     bfs=bfs,
    607                                     pivot_rule=pivot_rule,
--> 608                                     feas_tol=feas_tol)
    609         i = i + 1
    610         if iteration_limit is not None and i >= iteration_limit:

~\.conda\envs\ENGRI_1101\lib\site-packages\gilp\simplex.py in _simplex_iteration(lp, bfs, pivot_rule, feas_tol)
    472                     'manual_select': user_input,
    473                     'manual': user_input}[pivot_rule]
--> 474             r,t,d = ratio_test(k)
    475         # Update
    476         x[k] = t
```

```
~\.conda\envs\ENGRI_1101\lib\site-packages\gilp\simplex.py in ratio_test(k)
    445             index r, minimum ratio t, and d from solving A_b*d = A_
k."""
    446             d = np.zeros((1,n))
--> 447             d[:,B] = solve(A[:,B], A[:,k])
    448             ratios = {i: x[i]/d[0][i] for i in B if d[0][i] > feas_
tol}
    449             if len(ratios) == 0:

~\.conda\envs\ENGRI_1101\lib\site-packages\scipy\linalg\basic.py in solve
(a, b, sym_pos, lower, overwrite_a, overwrite_b, debug, check_finite, assum
e_a, transposed)
    212                                                 (a1, b1))
    213         lu, ipvt, info = getrf(a1, overwrite_a=overwrite_a)
--> 214         _solve_check(n, info)
    215         x, info = getrs(lu, ipvt, b1,
    216                         trans=trans, overwrite_b=overwrite_b)

~\.conda\envs\ENGRI_1101\lib\site-packages\scipy\linalg\basic.py in _solve_
check(n, info, lamch, rcond)
     27                          '.'.format(-info))
     28     elif 0 < info:
---> 29         raise LinAlgError('Matrix is singular.')
     30
     31     if lamch is None:

LinAlgError: Matrix is singular.
```

In [89]:

```
# Look at the first 3 iterations
nodes[0].show()
nodes[1].show()
nodes[2].show()
```

Let's fast-forward to the final iteration of the branch and bound algorithm.

In [90]:
```
nodes[-1].show()
```

**Q43:** Consider the feasible region that looks like a rectangular box with one corner point at the origin. What node does it correspond to in the tree? What is the optimal solution at that node?

**A:** it corresponds to the node with the objective value of 34. The optimal solution at this node is indeed (2,3), which is why it can be fathomed.

**Q44:** How many branch and bound nodes did we explore? What was the optimal solution? How many branch and bound nodes would we have explored if we knew the value of the optimal solution before starting branch and bound?

**A:** We ended up exploring nine nodes and would have explored 5 if we knew the value of the optimal solution beforehand.

# Bonus: Branch and Bound for Knapsack

Consider the following example:

| item | value | weight |
|---|---|---|
| 1 | 2 | 1 |
| 2 | 9 | 3 |
| 3 | 6 | 2 |

The linear program formulation will be:

$$\begin{aligned} \max \quad & 2x_1 + 9x_2 + 6x_3 \\ \text{s.t.} \quad & 1x_1 + 3x_2 + 2x_3 \leq 10 \\ & x_1, x_2, x_3 \geq 0, \quad \text{integer} \end{aligned}$$

In gilp, we can define this lp as follows:

In [91]:

```
lp = gilp.LP([[1,3,2]],
             [10],
             [2,9,6])

for fig in gilp.bnb_visual(lp):
    fig.show()
```