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 [1]:
```
# 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: A node can be fathomed if its optimal value is less optimal than another feasible solution which has already been found or if it is the same as a value which we have already found and we can determine it will not get any better. For example, if it is a minimization problem and you have found a bound that is higher than the current bound, this node can be fathomed.

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: In a maximization integer problem, the LP optimal value must be >= to the integer problem and in a minimization problem the LP optimal value must be <= the IP

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 if the current incumbent is 44, you are looking for paths towards solutions of 45 and higher.

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: This is true because and integer solution is feasible for an LP relaxation and if the input is integer then the linear relaxation is feasible for the IP, meaning the two must be equal.

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

A: This is true because an LP solution has the ability to be more precise than the IP solution (e.g in a maximization problem LPopt>=IPopt), so if the LP is infeasible, this means any integer solution (which is more constrained) must also be infeasible. In other words, since the IP solution is feasible for the LP solution, if the LP solution is infeasible, the IP solution must also be infeasible.

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: The IP is a maximization problem because both branches have z values greater than the root, and for the branch with z=17.8, the tree is continued to find z=20 > z=15.4

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 is no better current incumbent, it is not integer, and as far as we know it is feasible for the LP

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 of these nodes must be >= the current optimal value z=16.3 because this is a maximization problem.

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, …, 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.
max

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

In [2]:
```
data = pd.read_csv('knapsack_data_1.csv', index_col=0)
data
```

Out[2]:

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:

In [3]:
```
# 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: It does not change the optimal solution to the integer program because it has a weight of 30 which is infeasible for the IP when W=18

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

A: The LP relaxation's optimal value changes in that it now has another restriction because it cannot take a fraction of item 7, for example, meaning that its optimal value may decrease.

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 [4]:
```
data['value per unit weight'] = (data['value'] / data['weight']).round(2)
data
```

Out[4]:

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: A reasonable heuristic would be to chose items by their highest value per weight: e.g item 1, then 2, then 3, etc. as long as it does not overwhelm the weight limit

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: x1=1, x4=1 50+14 = 64

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 [5]:
```
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 and weight
        capcity (int): An integer-capacity for the knapsack
        integer (bool): True if the variables should be integer. False otherwise.
    """
    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 capacity
    # 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 [6]:
```
# 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 [7]:
```
m, x = Knapsack(data, 18)
```

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

In [73]:
```
solve(m)
```

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

Out[73]:
```
{'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 what was found with the heuristic

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 because it is higher than the current incumbent. The new constraints should be x2 >= 1 or x2 <=0

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 [9]:
```
m, x = Knapsack(data, 18)
m.Add(x[2] <= 0)
solve(m)
```

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

Out[9]:
```
{'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 [10]:
```
m, x = Knapsack(data, 18)
m.Add(x[2] >= 1)
solve(m)
```

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

Out[10]:
```
{'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 can be fathomed because we already have an incumbent with value 64>60.

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 there is already an incument of 64 and this means that we are looking for solutions of 65 and higher

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 [14]:
```
# 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)



solve(m)
# node with z= 51 is fathomed because current incumbent is better
```

```
Objective = 65.42857142857143
iterations : 0
branch-and-bound nodes : 0
```

Out[14]:
```
{'x_1': 1.0,
 'x_2': 0.0,
 'x_3': 0.0,
 'x_4': 1.0,
 'x_5': 0.0,
 'x_6': 0.14285714285714302}
```

In [15]:
```
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)
#node fathomed bc 44<64
```

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

Out[15]:
```
{'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 [16]:
```
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)
```

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

Out[16]:
```
{'x_1': 1.0, 'x_2': 0.0, 'x_3': 0.0, 'x_4': 1.0, 'x_5': 0.0, 'x_6': 0.0}
```

A: Optimal solution in x1 = 1, x2=0, x3=0, x4=1, x5=0, x6=0 obj value = 64 because there are no other nodes to explore and this is the best integer value

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

A: 11

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

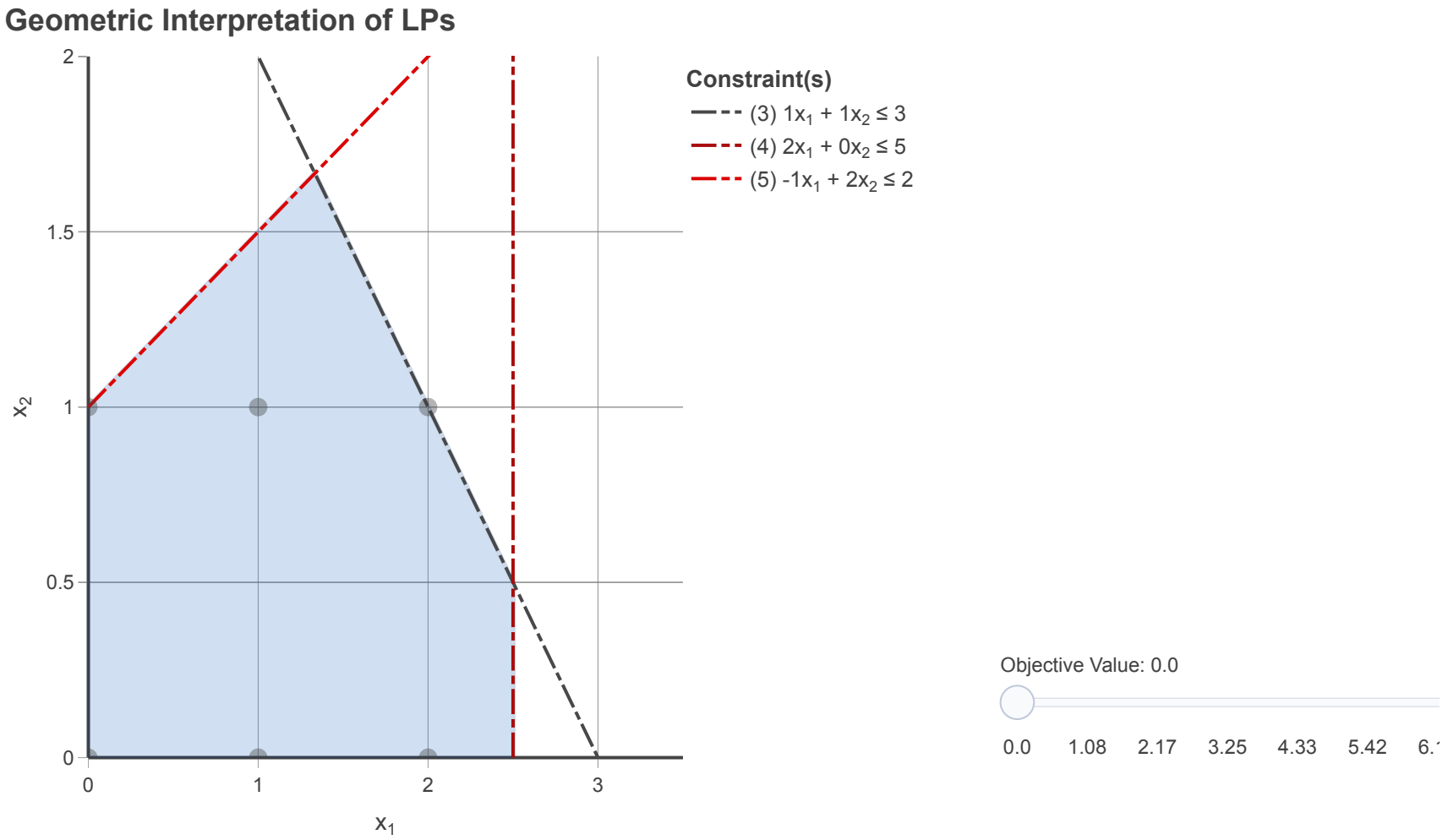
Part 3: Cutting Planes

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

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

In [17]:
```
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
```



Processing math: 32%

Q22: List every feasible solution to the integer program.

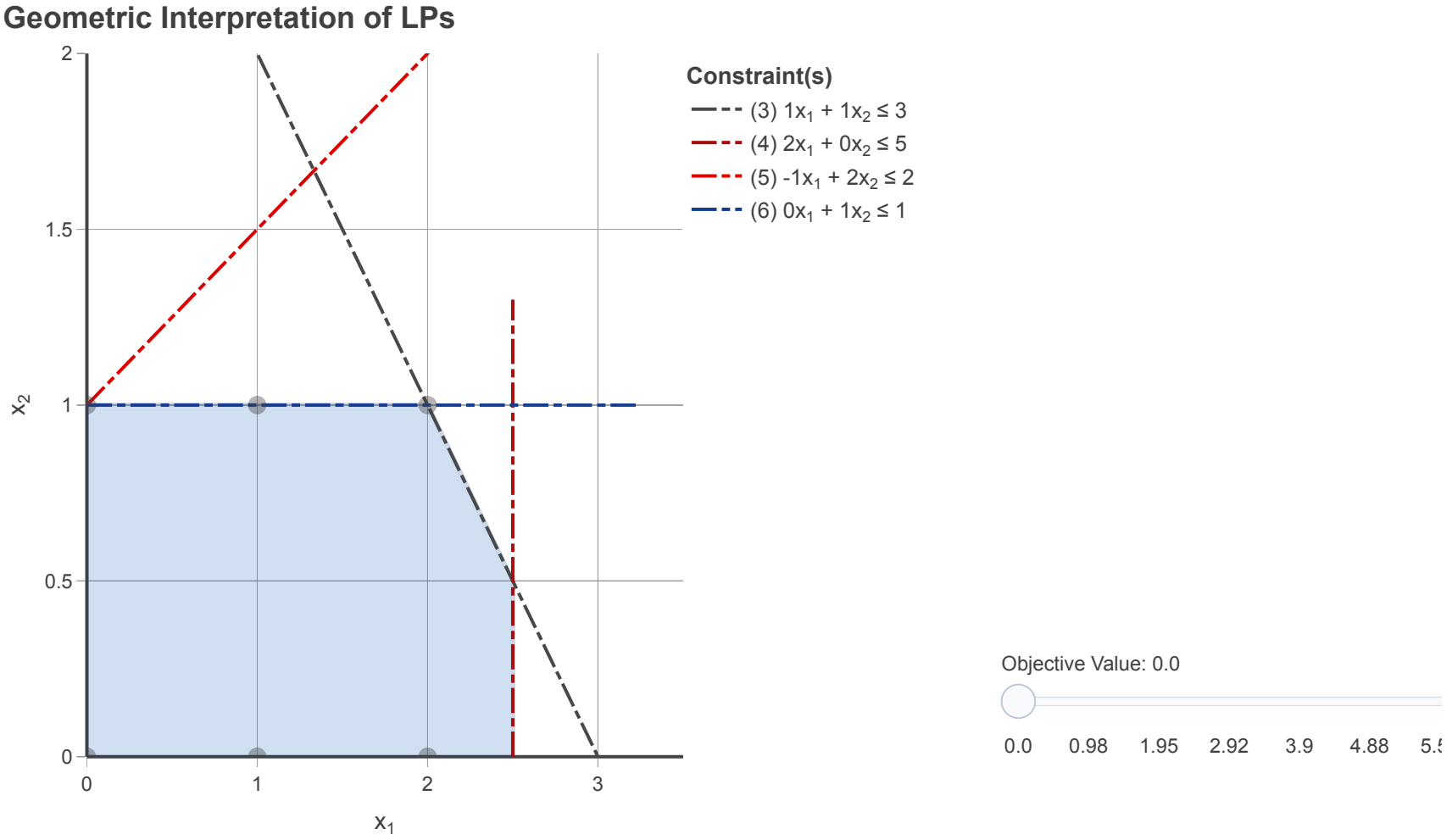
A: (0,0) Obj = 0, (1,0) Obj= 2, (2,0) Obj= 4, (0,1) Obj = 1, (1,1) Obj = 3, (2,1) Obj = 5

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 because all feasible points for the integer program remain feasible with this constraint, while it would limit the amount of feasible points for the LP

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

In [18]:
```
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?

Processing math: 32%

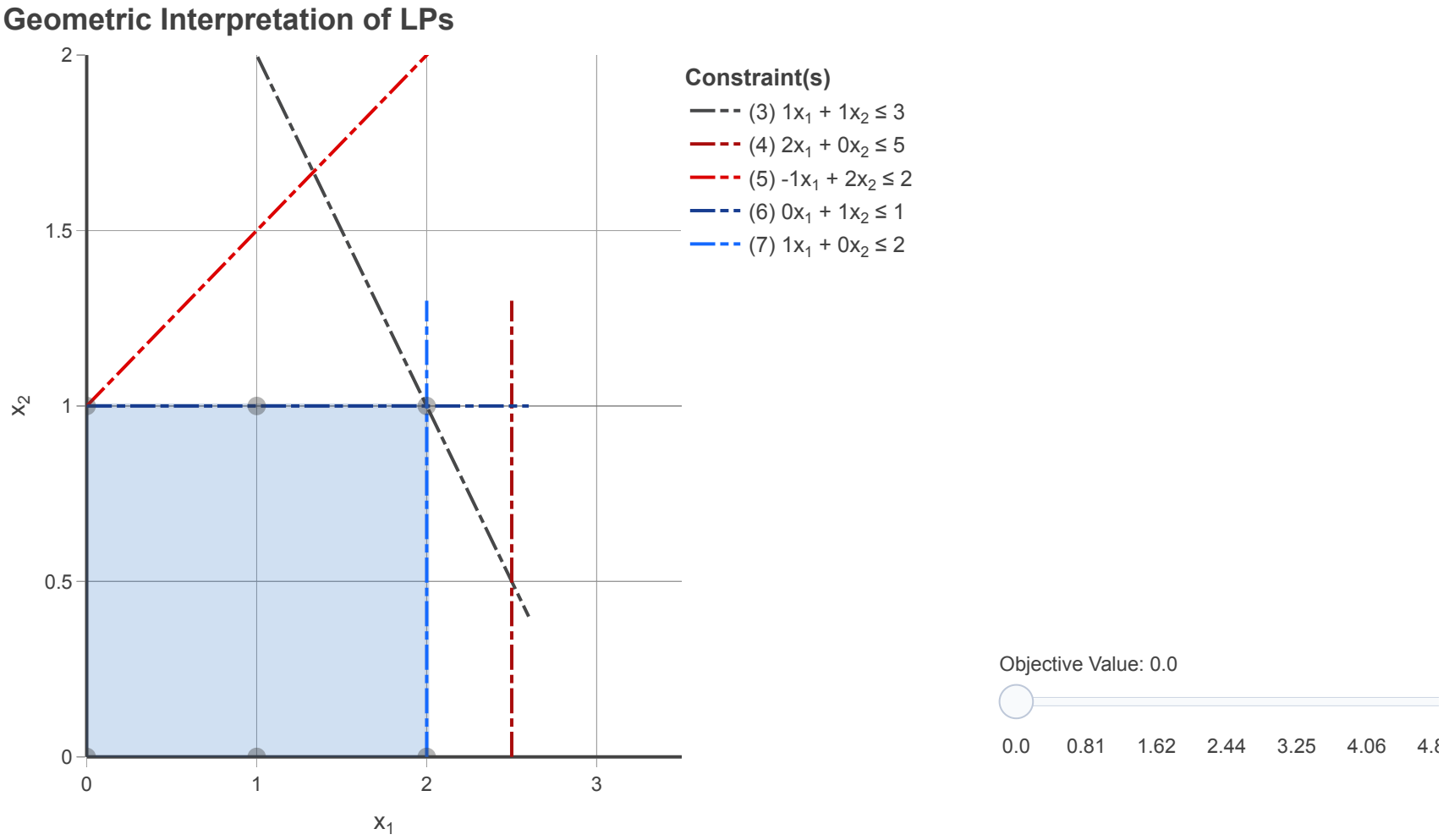
A: No because it does not limit the number of feasible LP solutions.

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

A: x1<=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 [19]:
```
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:

Processing math: 32%

In [20]:
```
data
```

Out[20]:

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: We can only take one of these items, otherwise we would surpass W = 18. x1 + x2 +x3 <= 1 would be a new constraint.

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, a feasible solution could have been x1 = .5, x2=.5, x3=.5 which would have weight 16<18.

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 no more than 2 of these items can be chosen and remain with a weight under 18.

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 [21]:
```
m, x = Knapsack(data, 18)
# TODO: Add cutting planes here
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[21]:
```
{'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 [27]:
```
# Template
m, x = Knapsack(data, 18)
# Add constraints here
m.Add(x[1] + x[2] + x[3] <= 1)
m.Add(x[4] + x[5] + x[6] <= 2)
m.Add(x[5]>=1)
m.Add(x[4]>=1)



solve(m)
# This node is fathomed bc 51<64
```

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

Out[27]:
```
{'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 [28]:
```
m, x = Knapsack(data, 18)
# Add constraints here
m.Add(x[1] + x[2] + x[3] <= 1)
m.Add(x[4] + x[5] + x[6] <= 2)
m.Add(x[5]>=1)
m.Add(x[4]<=0)


solve(m)
#This node is fathomed bc we are looking for solutions >=65
```

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

Out[28]:
```
{'x_1': 1.0,
 'x_2': 0.0,
 'x_3': 0.0,
 'x_4': 0.0,
 'x_5': 1.0,
 'x_6': 0.28571428571428586}
```

Processing math: 32%

In [ ]:

A: Solution has objective value 64, as was found previously

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

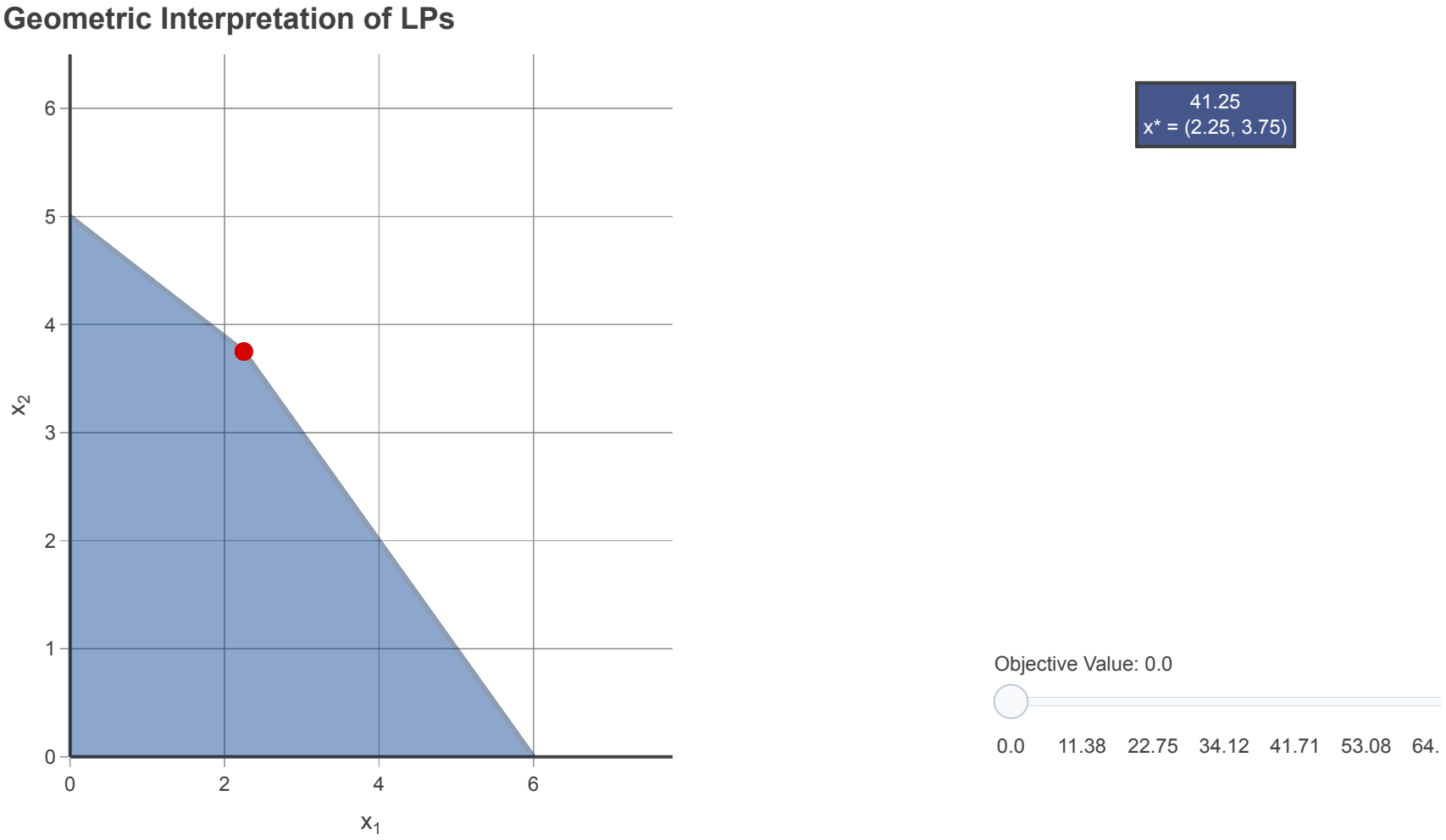
A: Yes, by exploring 5 nodes, 6 fewer than before.

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{align*} \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{align*}

In [29]:
```
nodes = gilp.bnb_visual(gilp.examples.STANDARD_2D_IP)
```

In [30]:
```
nodes[0].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: (2.25, 3.75) Obj = 41.25

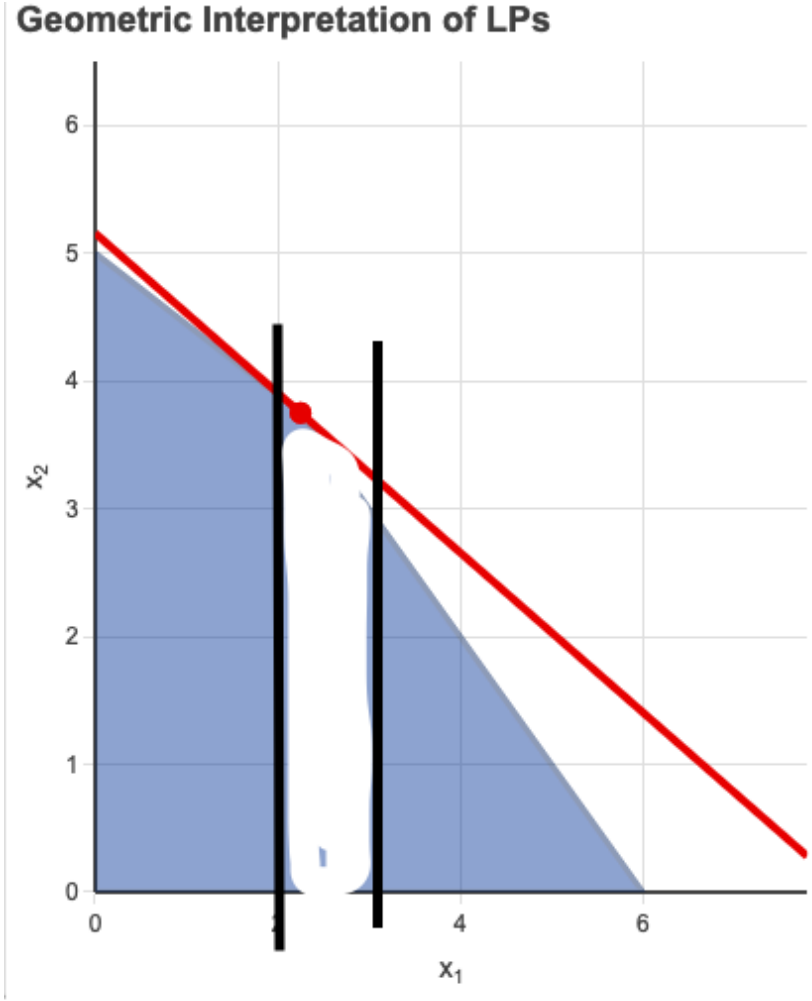
Q33: Assume that we always choose the variable with the minimum index to branch on if there are multiple options. Write down (in full) each of the LPs we get after branching off the root node.

A: max 5x1 + 8x2 s.t x1 + x2 <= 6 5x1 + 9x2 <=45 x1,x2 >= 0 x1<= 2 "or"

```
max 5x1 + 8x2
s.t x1 + x2 <= 6
    5x1 + 9x2 <=45
    x1,x2 >= 0
    x1 >= 3
```

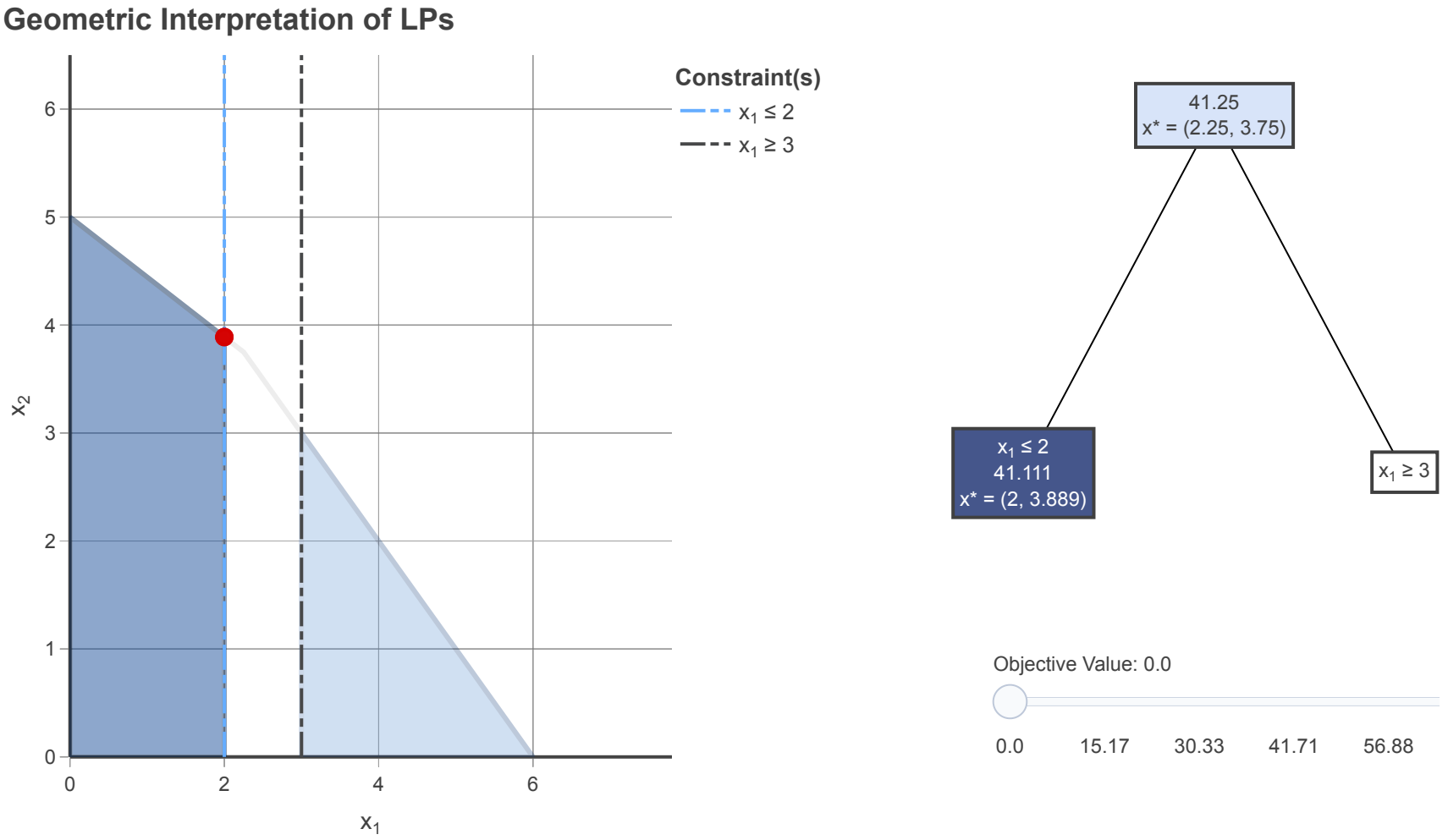
Q34: Draw the feasible region to each of the LPs from **Q33** on the same picture.

A:



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

In [31]:
```
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.

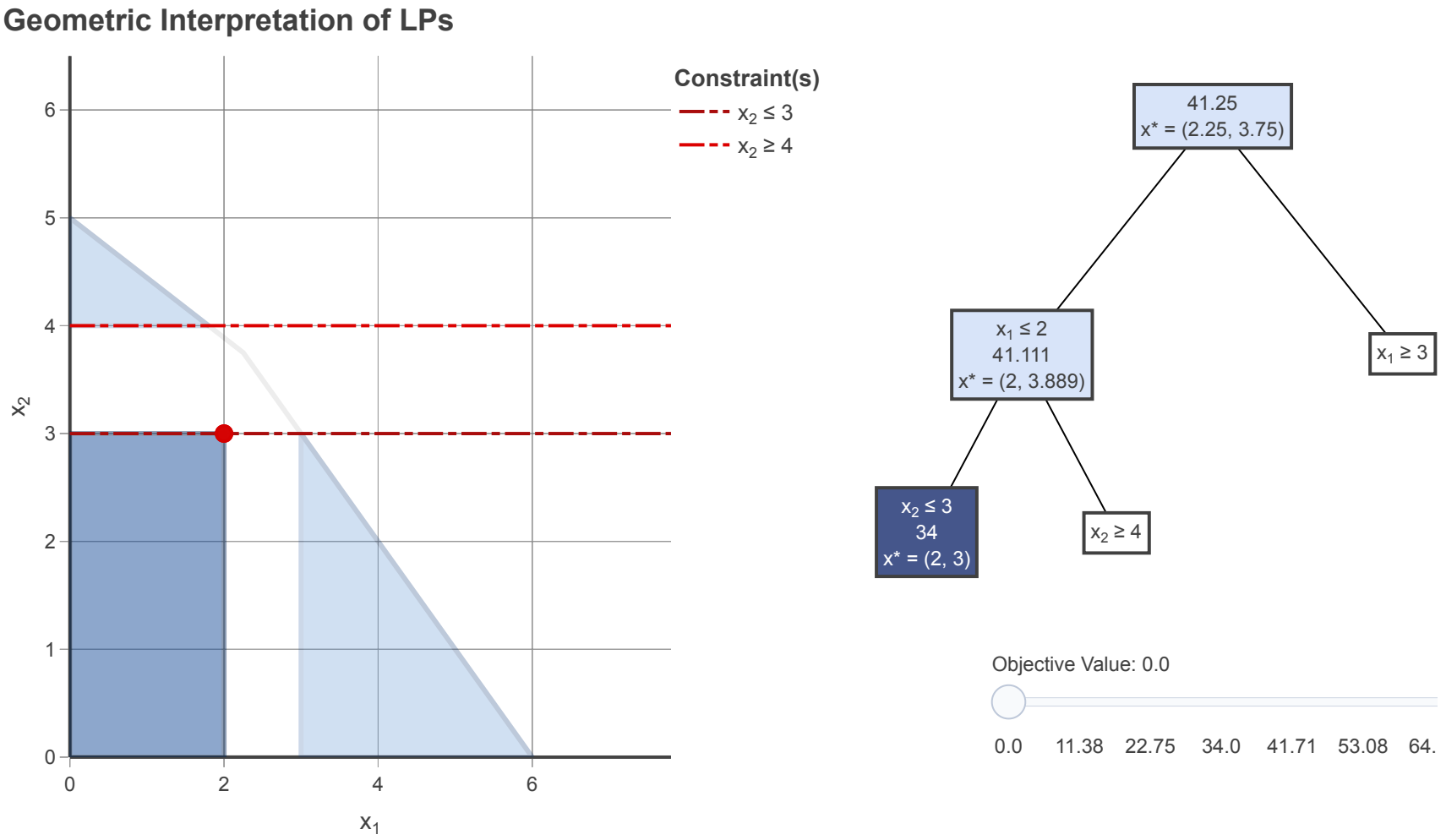
Q35: Which feasible solutions to the LP relaxation are removed by this branch?

A: any where 2<x1<3

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

A: x2<=3 x2>=4. It will be broken into 3 pieces

In [32]:
```
nodes[2].show()
```



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

A: (2,3) Obj= 34. We need not explore further because the solution is integer

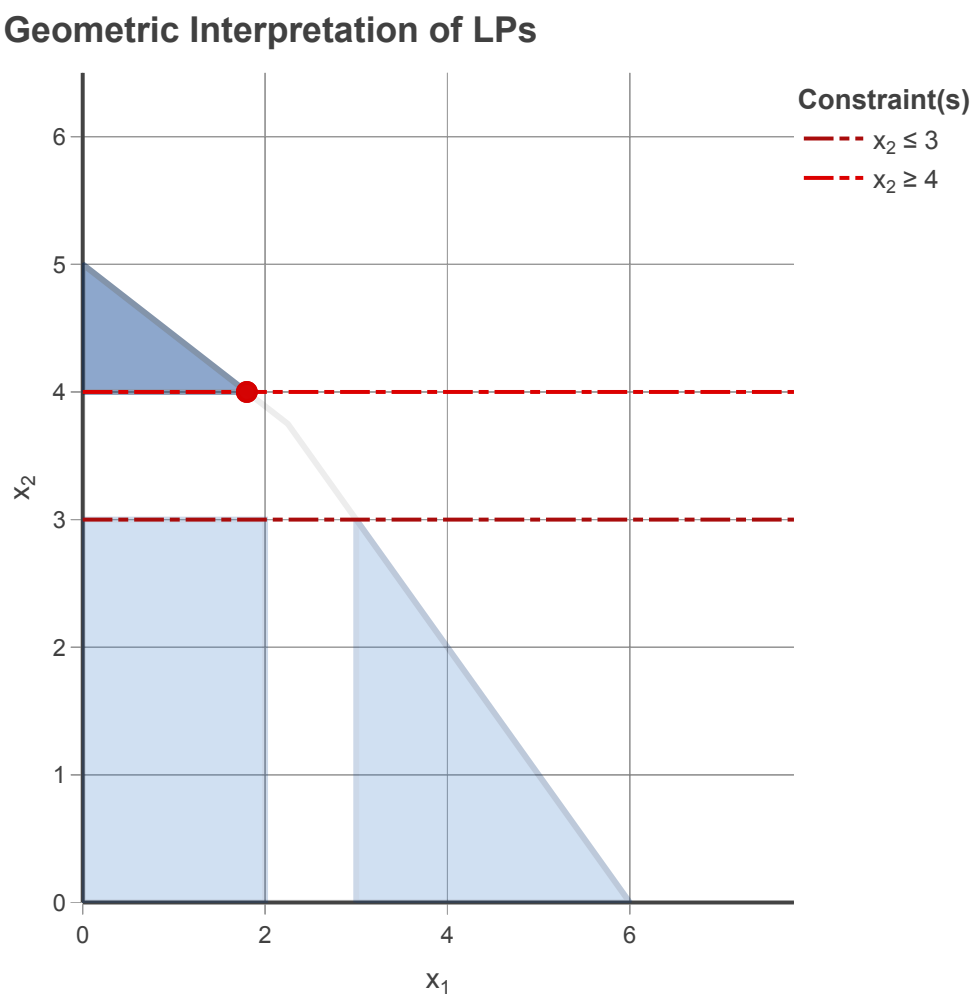
Q38: Recall shaded nodes have been explored and the node shaded darker (and feasible region shaded darker) correspond to the current node and its feasible region. Nodes not shaded have not been explored. How many nodes have not yet been explored?

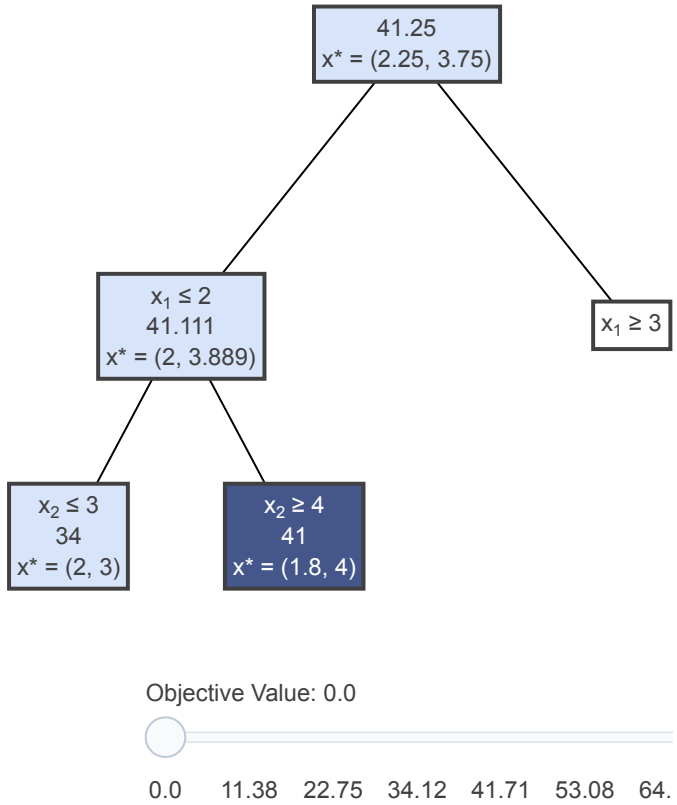
A: 2

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?

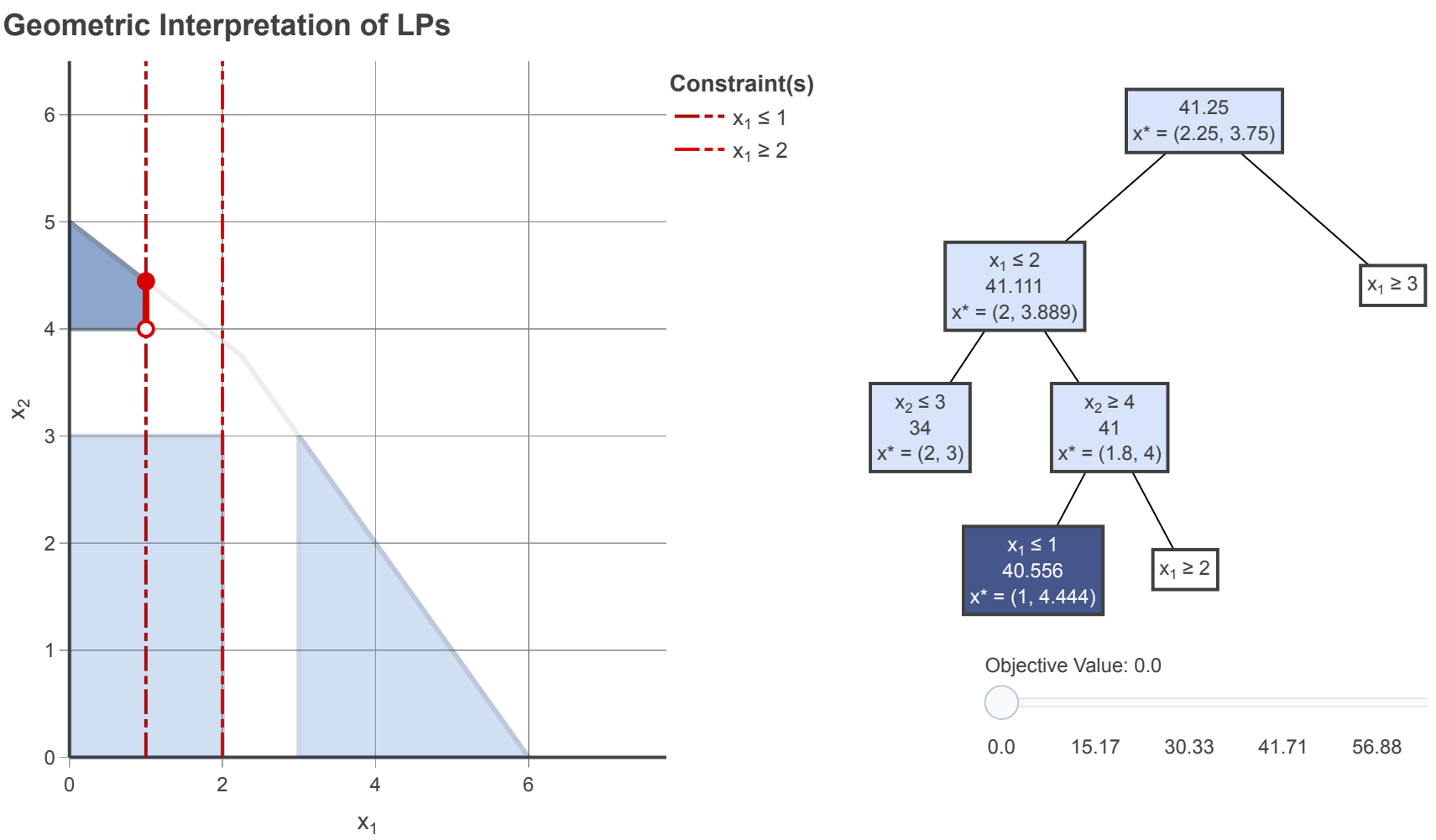
A: 3. The leaf nodes represent the remaining feasible regions.

In [33]:
```
# Show the next two iterations of the branch and bound algorithm
nodes[3].show()
nodes[4].show()
```





Processing math: 32%

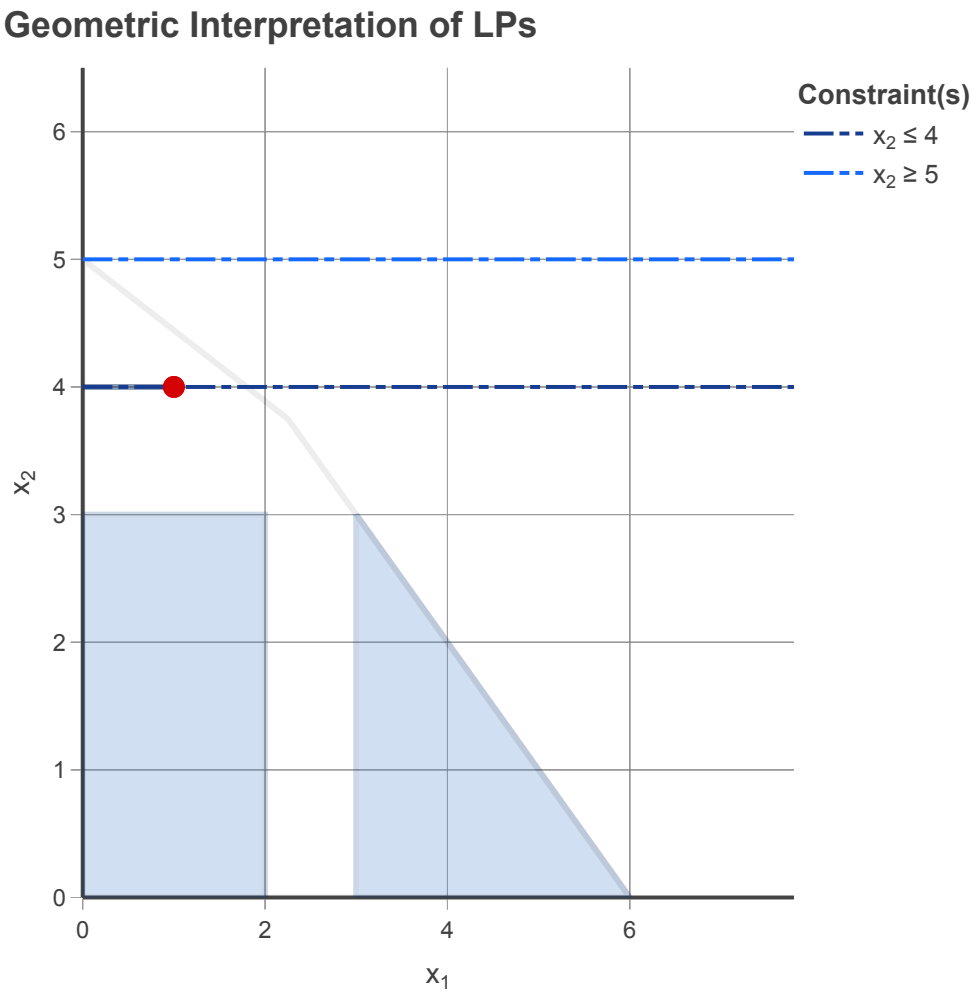


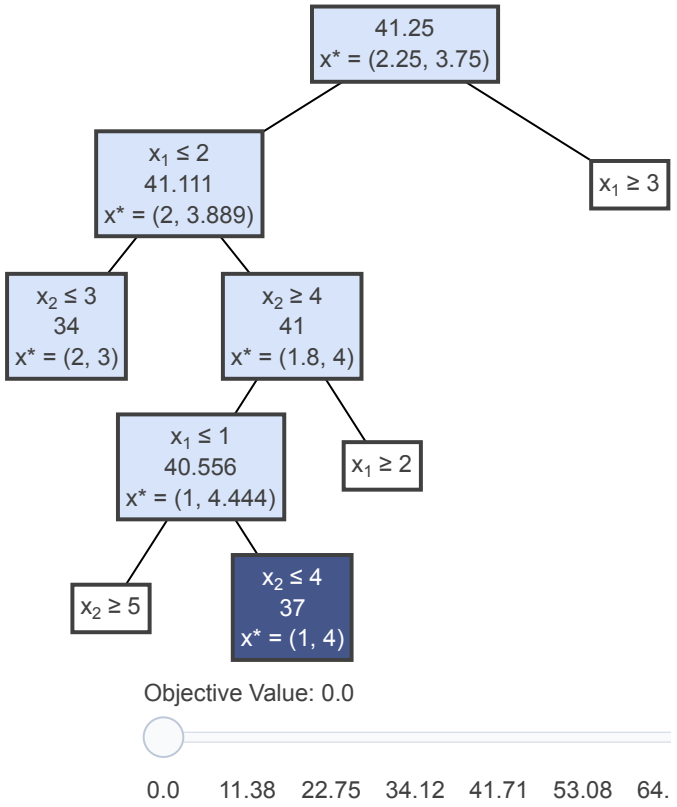
Q40: At the current (dark) node, we added the constraint x_1 \leq 1. Why were the fractional solutions 1 < x_1 < 2 not eliminated for x_2 <= 3?

A:Because this path of the tree only includes x1<= 2 and x2>=4.

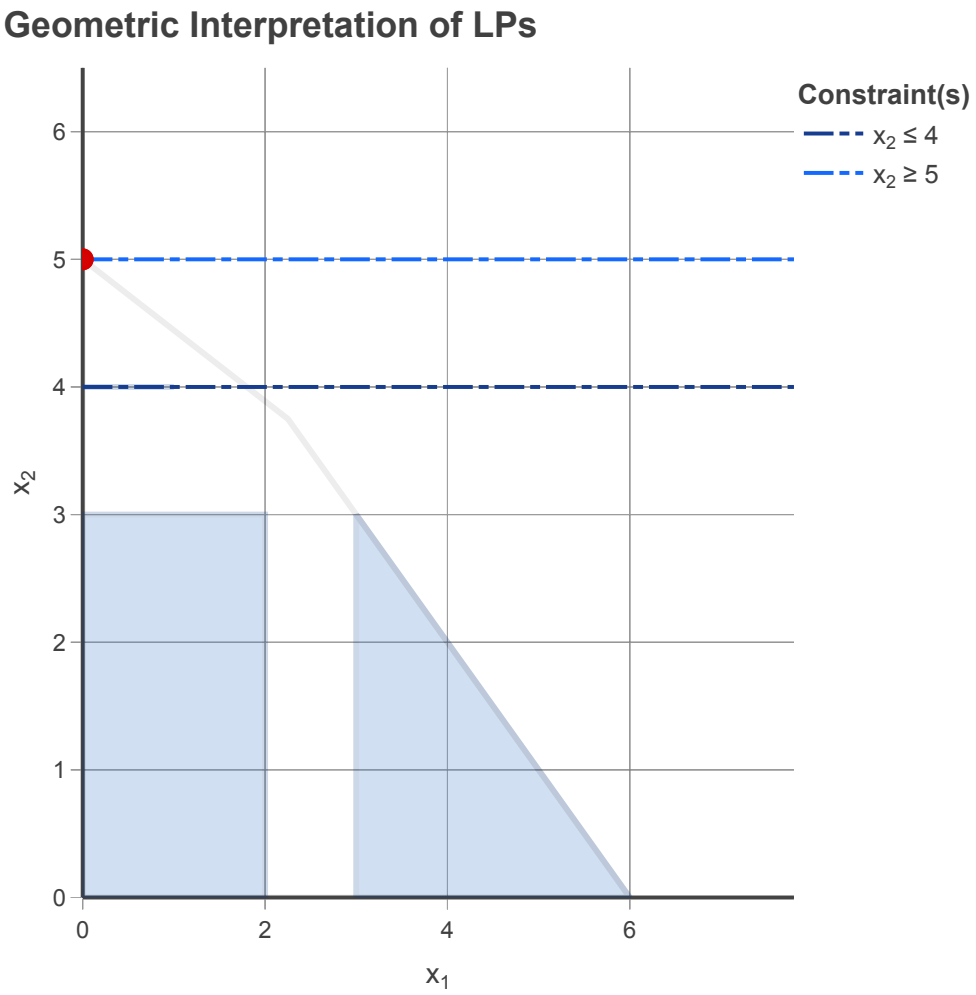
Processing math: 32%

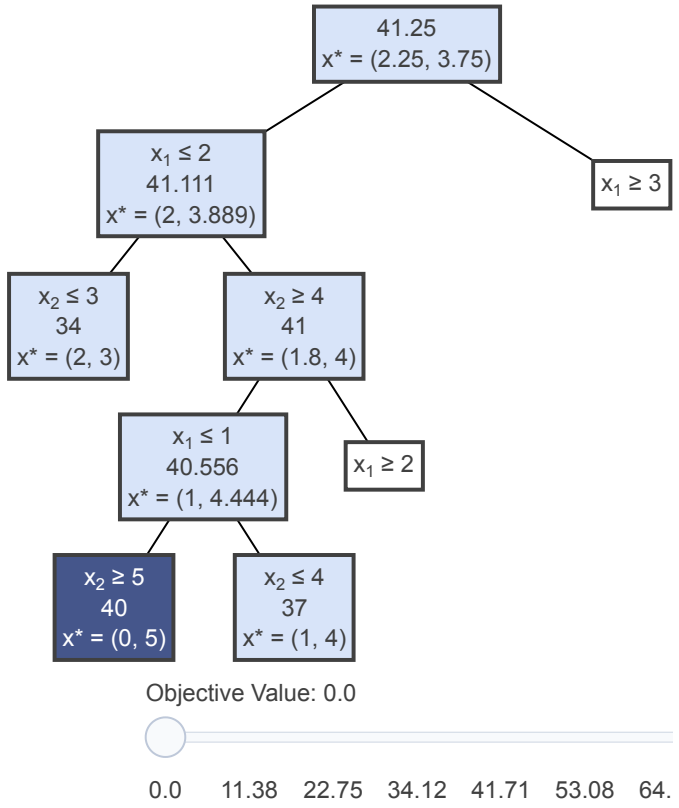
In [34]:
```
# Show the next three iterations of the branch and bound algorithm
nodes[5].show()
nodes[6].show()
nodes[7].show()
```



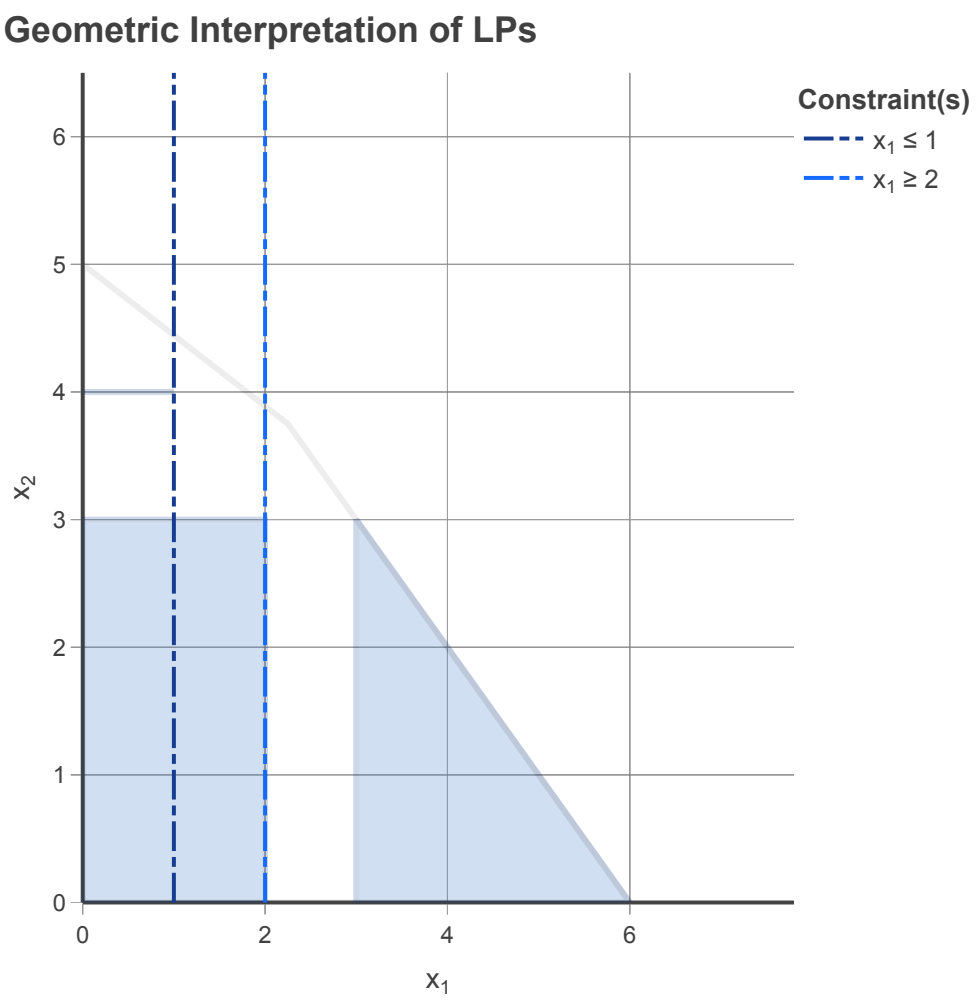


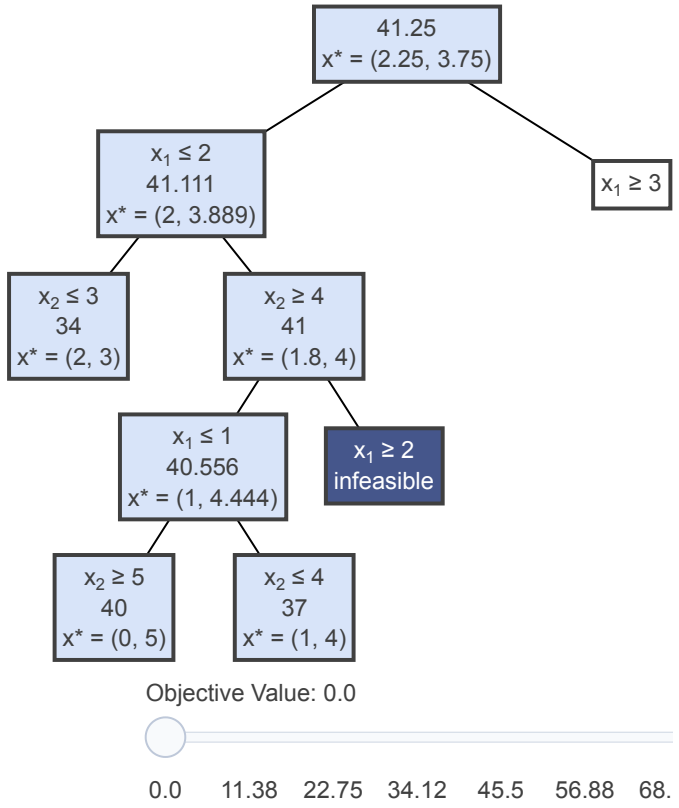
Processing math: 32%





Processing math: 32%

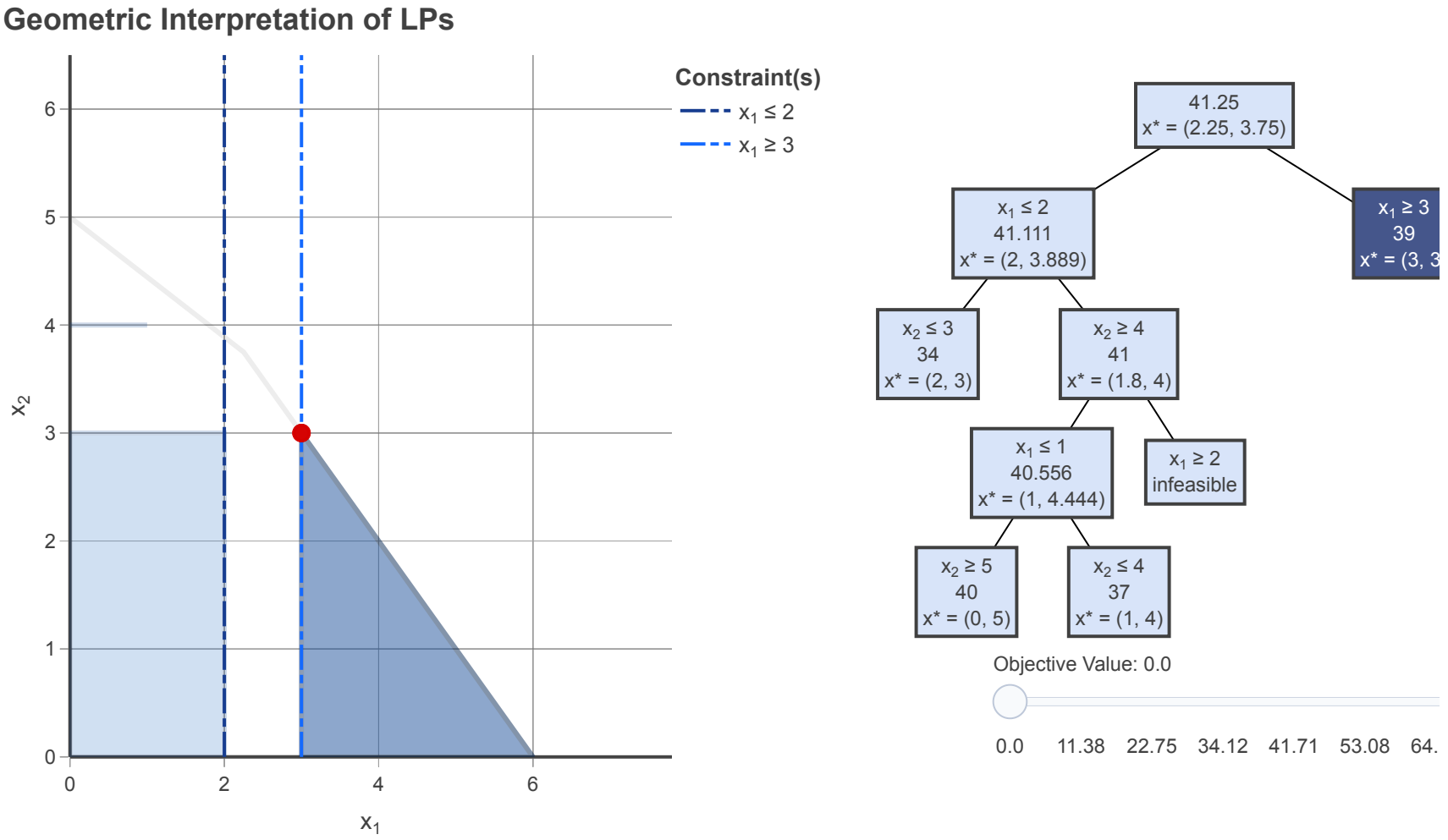




Q41: What constraints are enforced at the current (dark) node? Why are there no feasible solutions at this node?

A: At the current dark node the followed constraints are x1<= 2 and x2>=4. There are no feasible solutions at this point because x1 cant be both <2 and >2 and at x1=2, x2<4.

In [35]:
```
nodes[8].show()
```



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

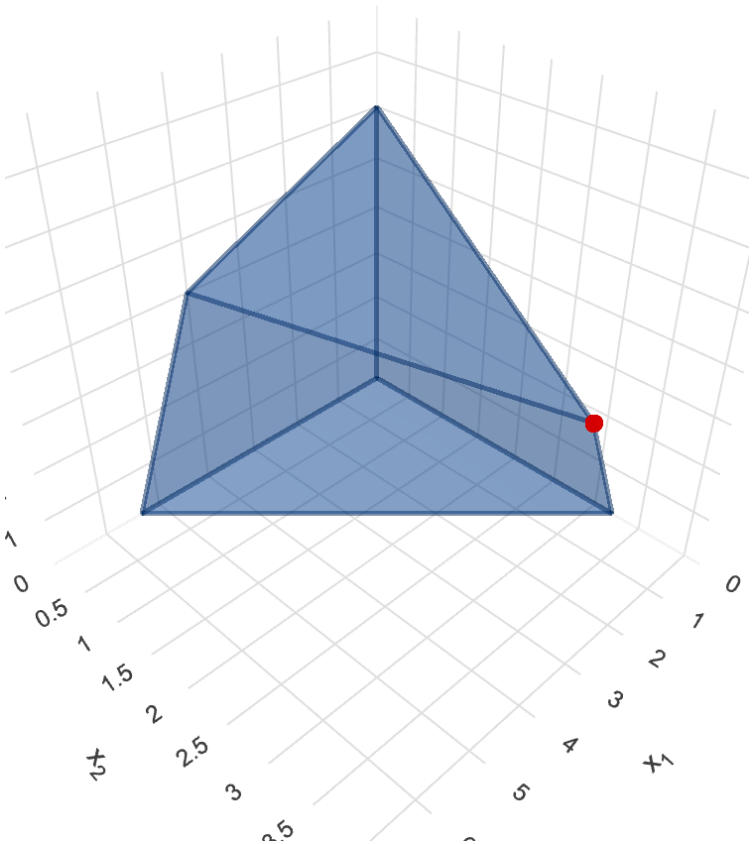
A: We are done. The nodes enforcing x2<=3, x1>= 3, x1>=2 and x2<=4 are fathomed because they are either infeasible or less than the current incumbent, where the optimal solution is shown to be = 40 with x = (0,5).

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

In [36]:
```
nodes = gilp.bnb_visual(gilp.examples.VARIED_BRANCHING_3D_IP)
```

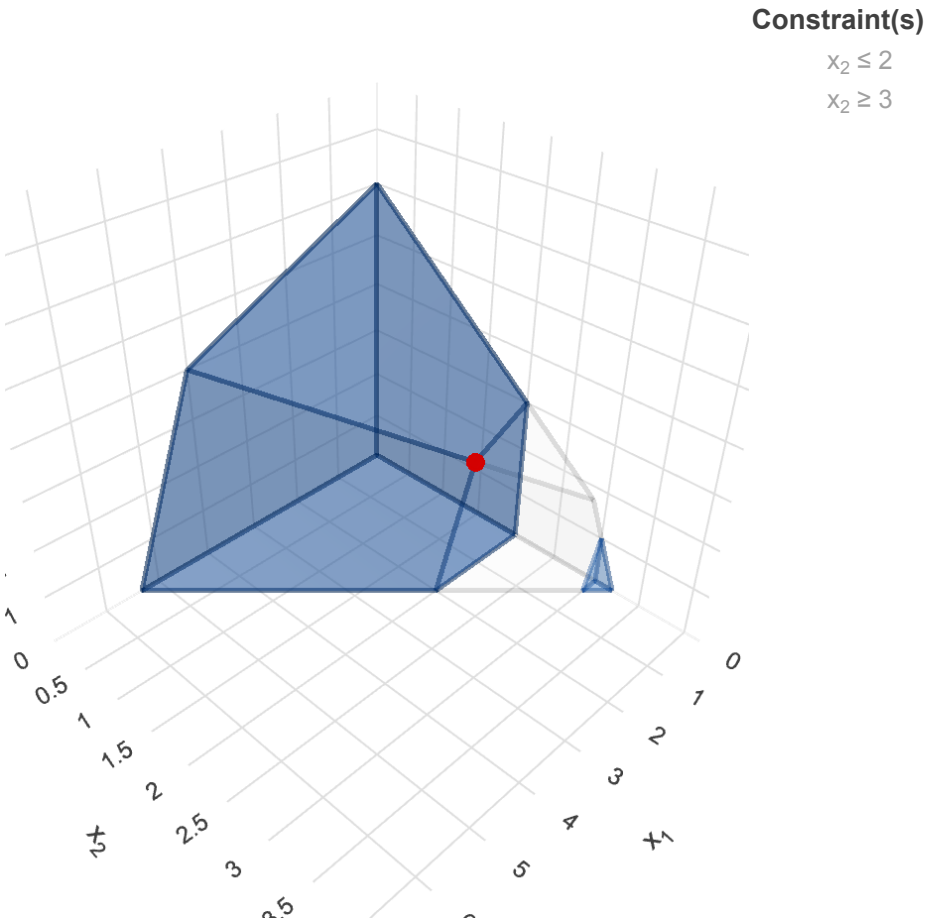
In [37]:
```
# Look at the first 3 iterations
nodes[0].show()
nodes[1].show()
nodes[2].show()
```

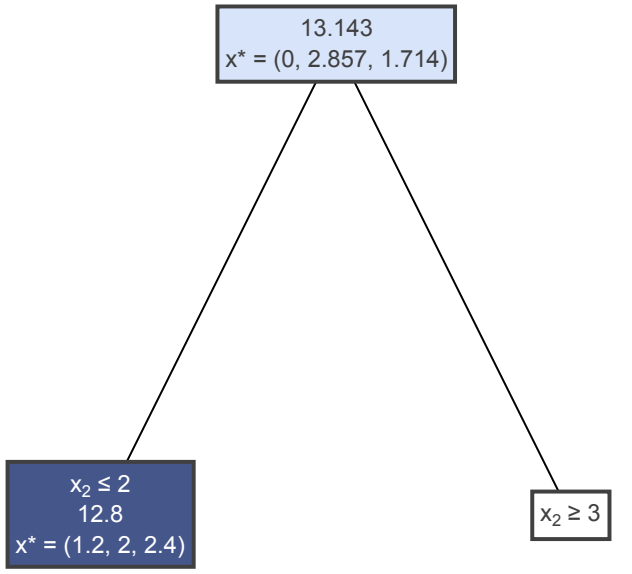
Geometric Interpretation of LPs



Processing math: 32%

Geometric Interpretation of LPs



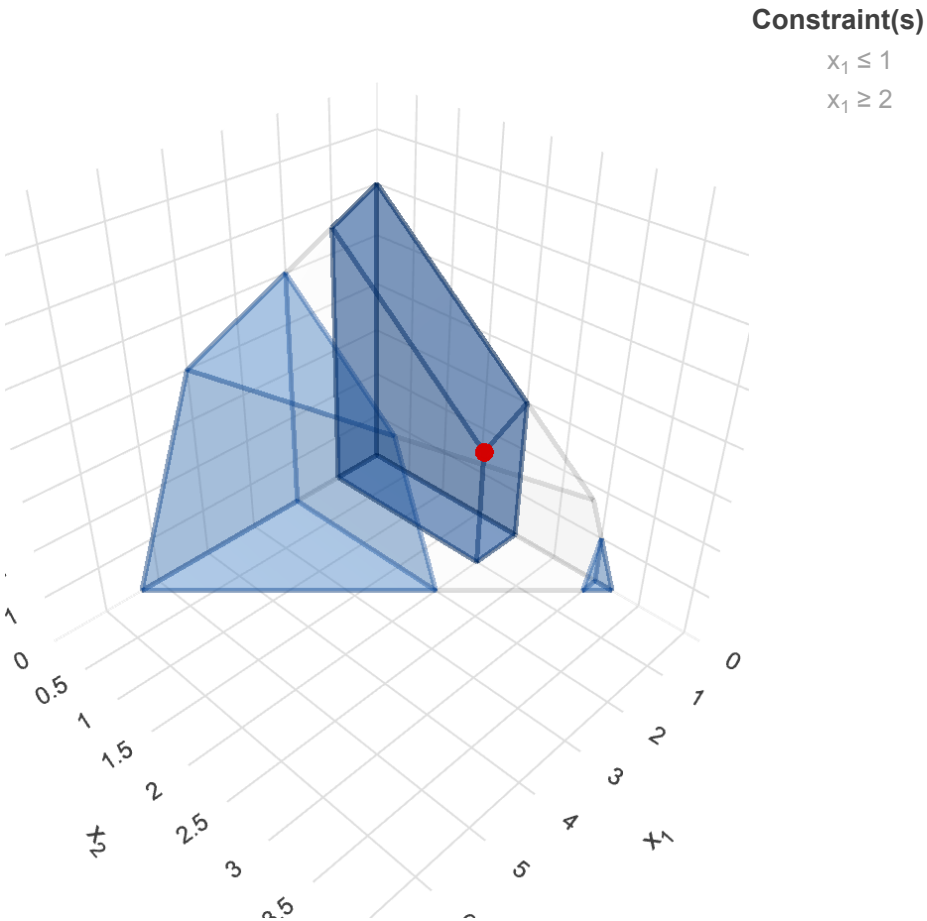


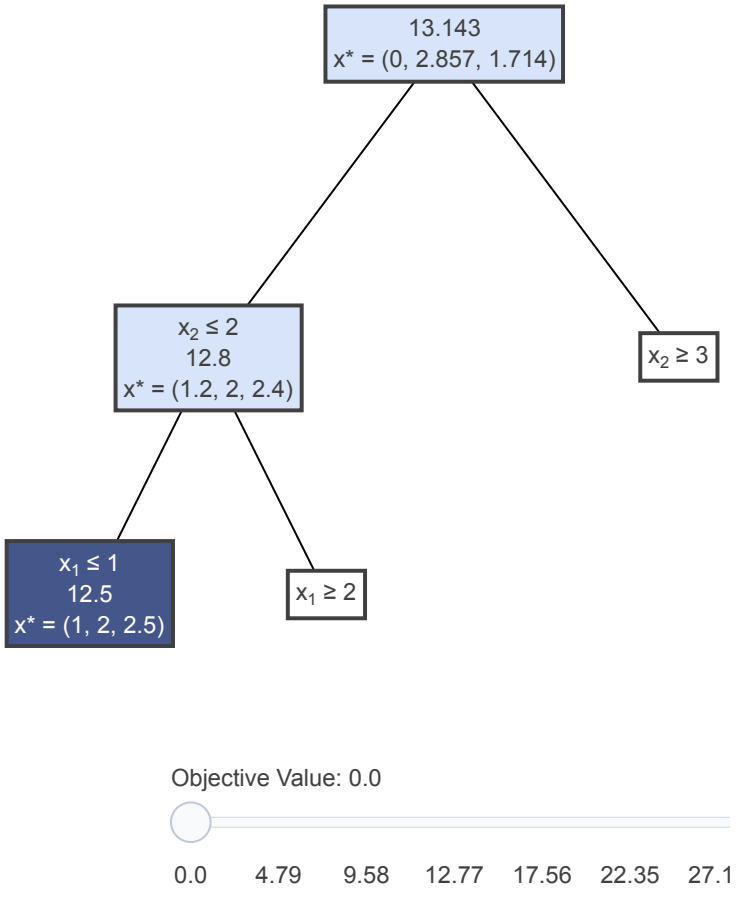
Objective Value: 0.0

0.0 4.79 9.58 12.8 17.56 22.35 27.1

Processing math: 32%

Geometric Interpretation of LPs



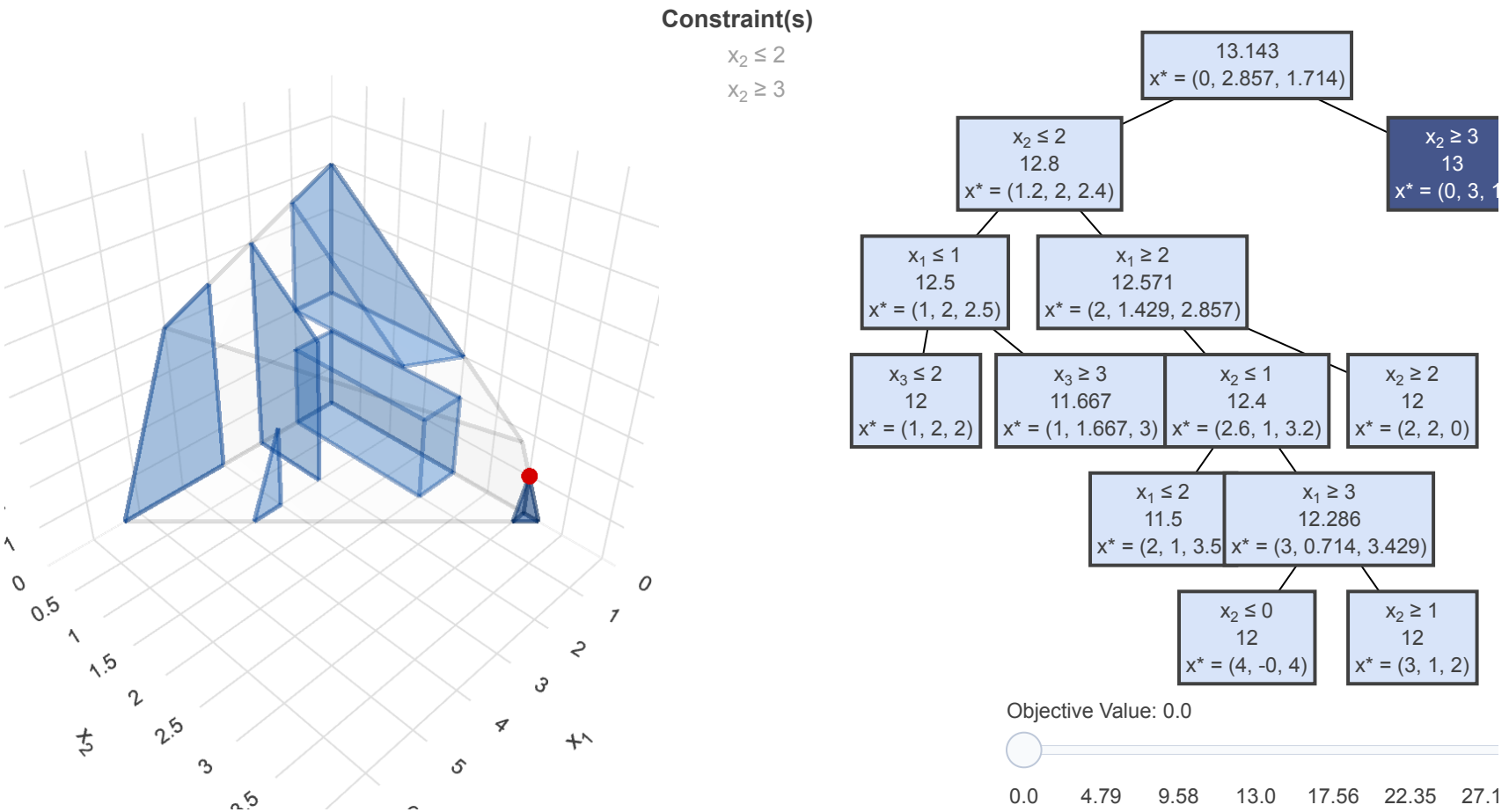


Let's fast-forward to the final iteration of the branch and bound algorithm.

Processing math: 32%

In [38]:
```
nodes[-1].show()
```

Geometric Interpretation of LPs



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 constraint x3<= 2 which has optimal solution 12.

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 explored 13 nodes, with the optimal solution 13. Had we known this before we would have only had to explore 2-3 nodes.

Bonus: Branch and Bound for Knapsack

Consider the following example:

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

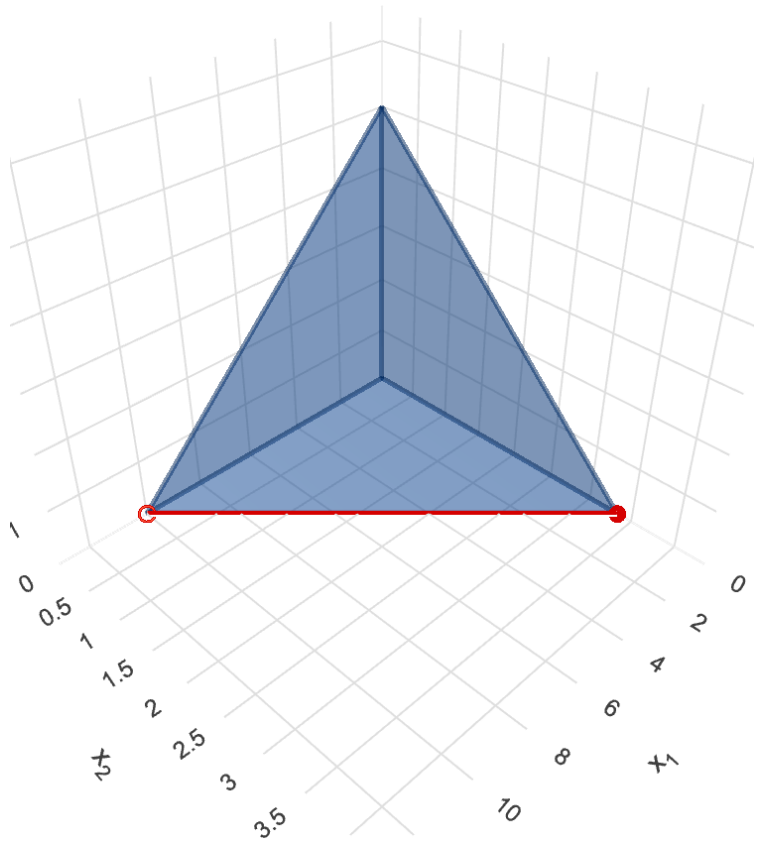
The linear program formulation will be:
$$\begin{align*} \max \quad & 2x_1+9x_2+6x_3\\ \text{s.t.} \quad & 1x_1+3x_2+2x_3 \leq 10 \\ & x_1, x_2, x_3 \geq 0, \quad \text{integer} \end{align*}$$

In gilp, we can define this lp as follows:

In [39]:
```
lp = gilp.LP([[1,3,2]],
             [10],
             [2,9,6])

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

Geometric Interpretation of LPs

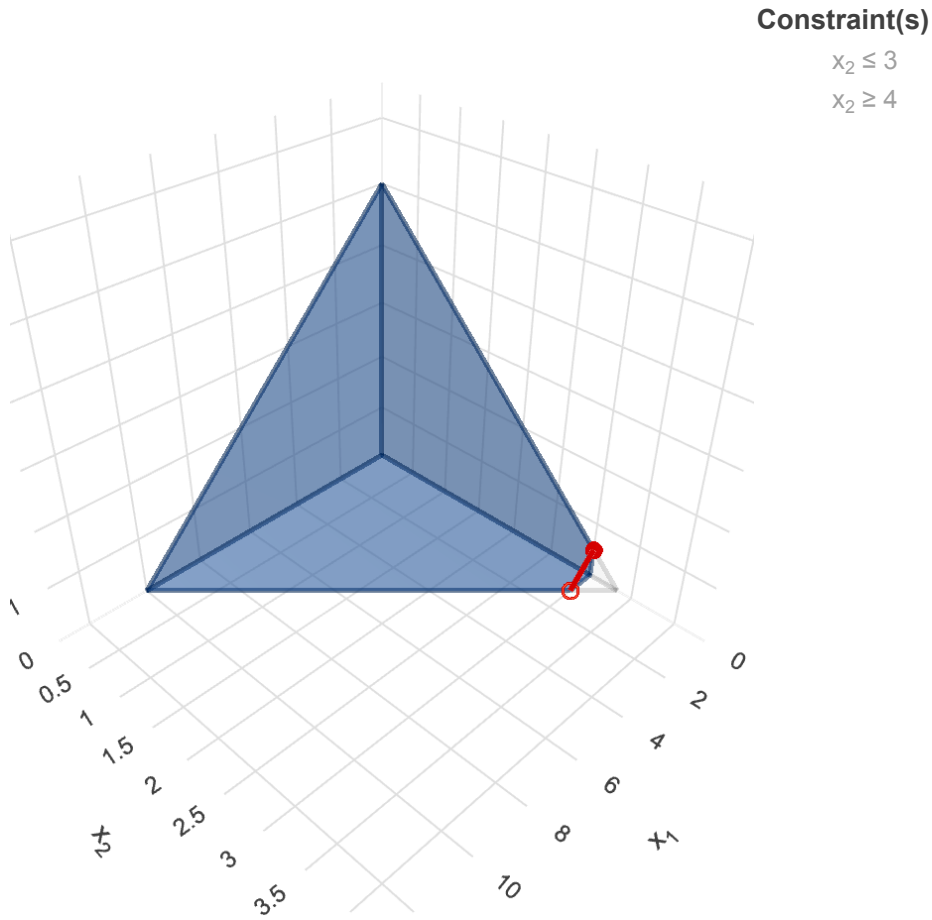


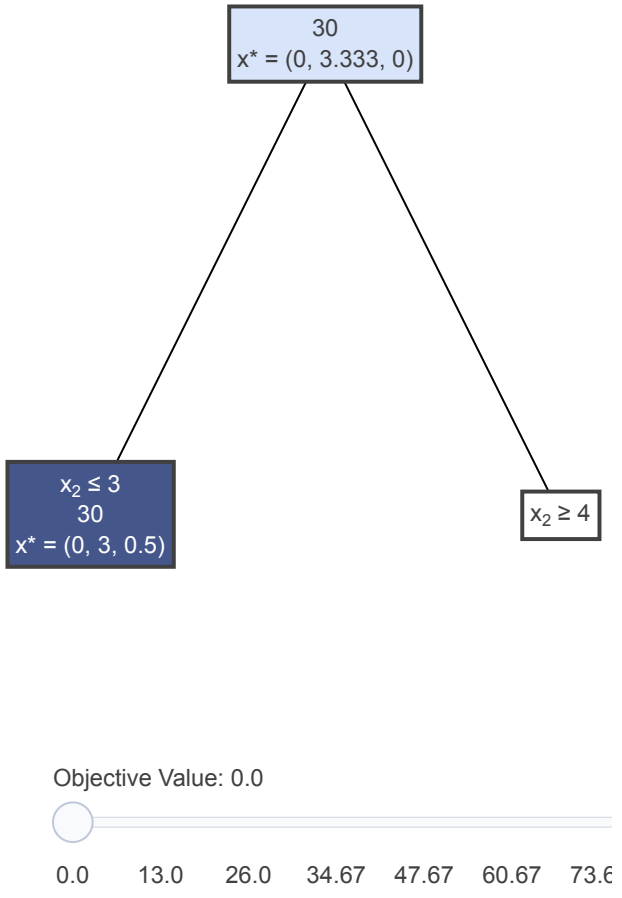
Objective Value: 0.0

0.0 13.0 26.0 34.67 47.67 60.67 73.6

Processing math: 32%

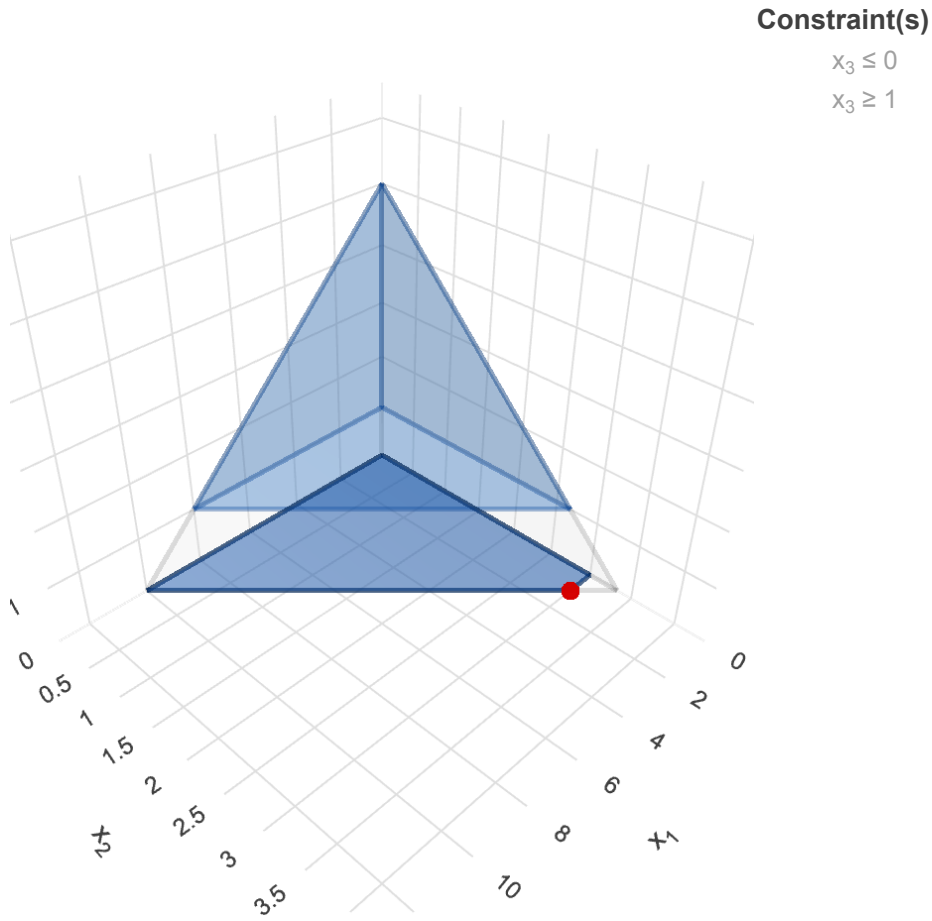
Geometric Interpretation of LPs

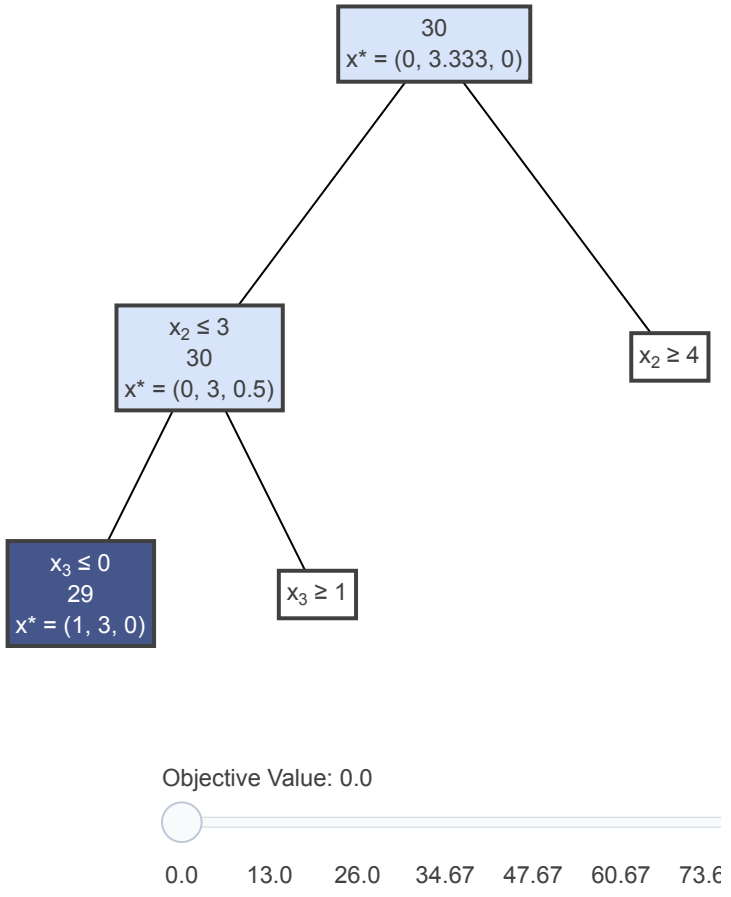




Processing math: 32%

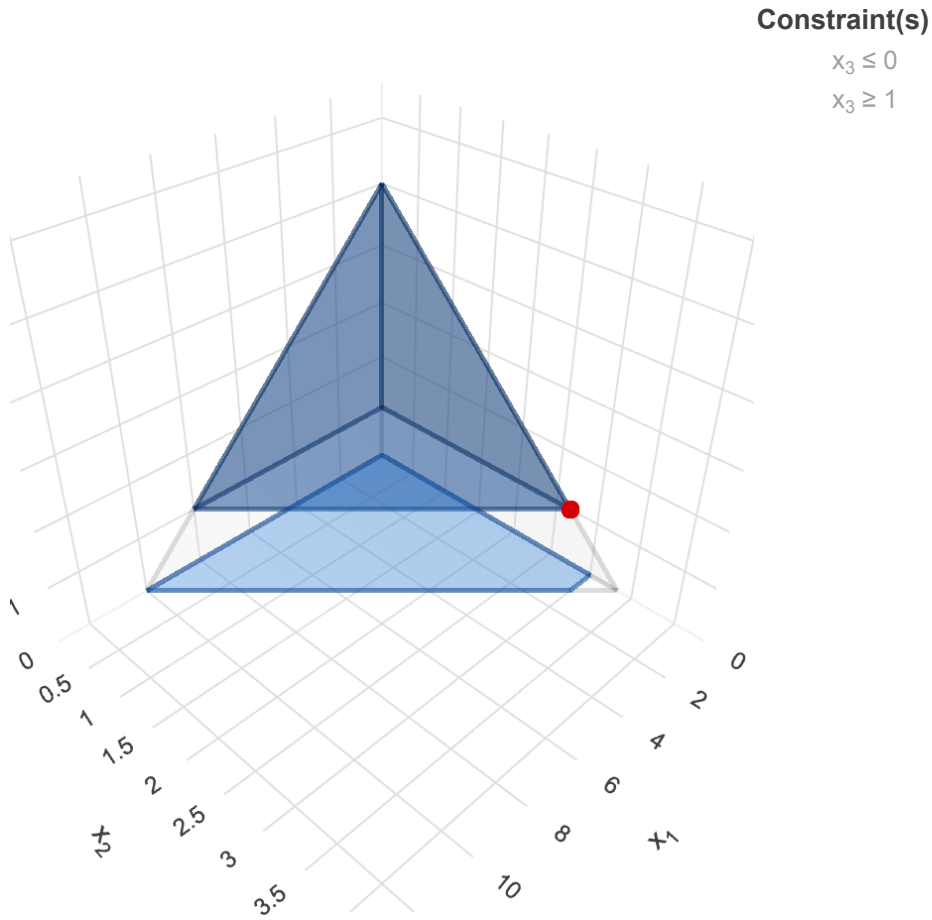
Geometric Interpretation of LPs

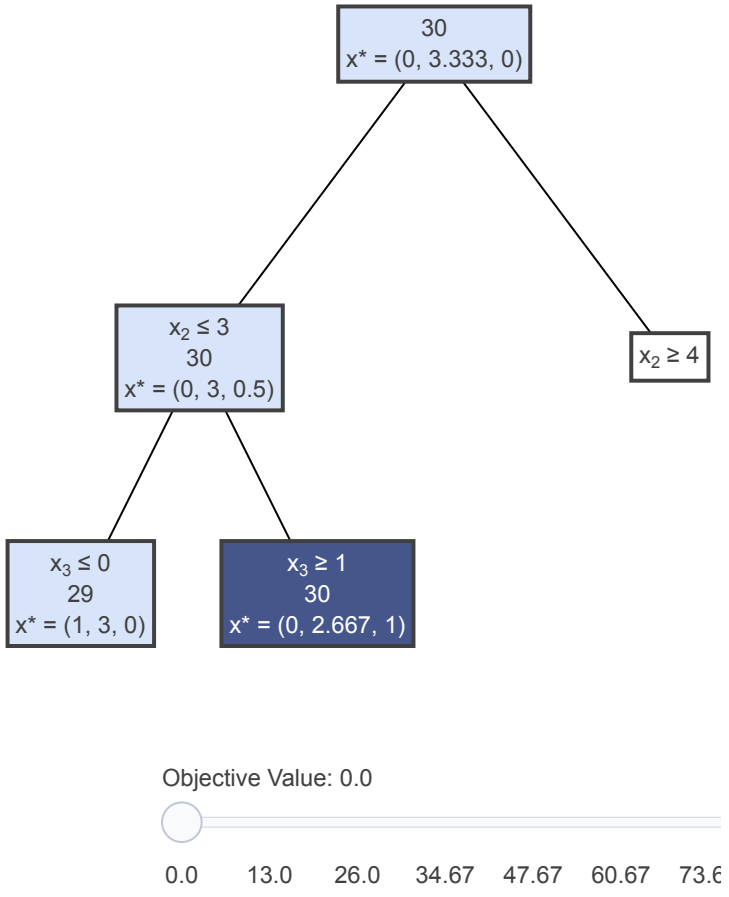




Processing math: 32%

Geometric Interpretation of LPs



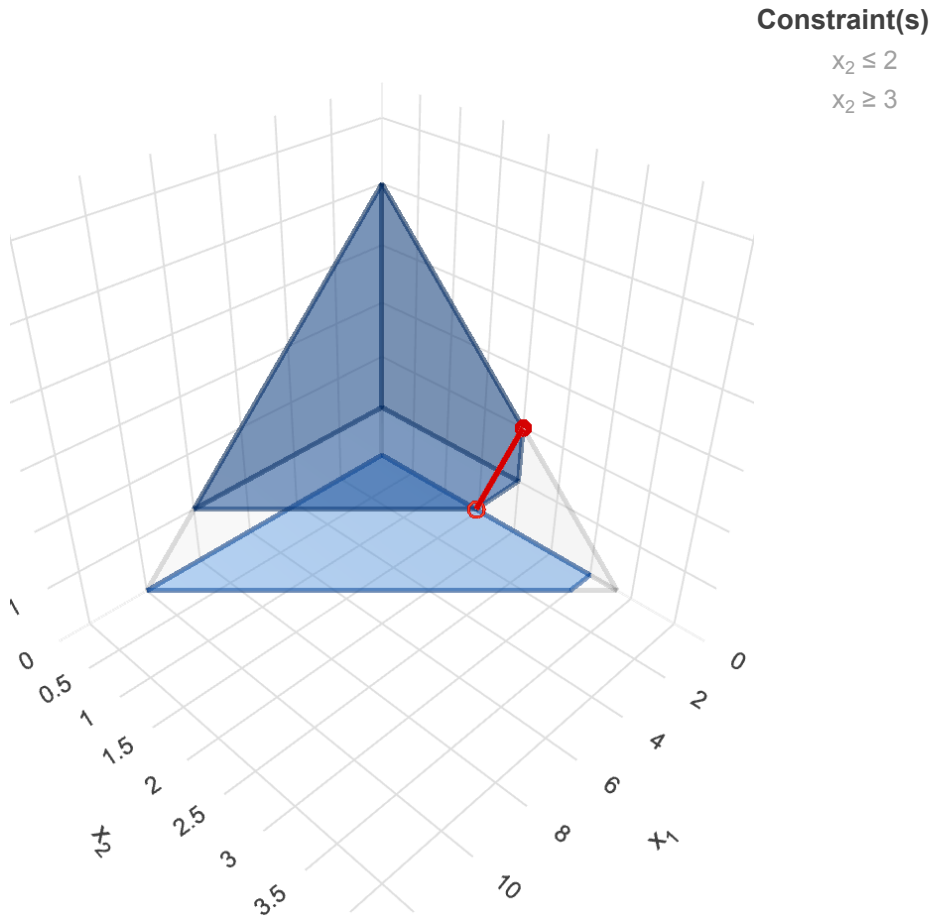


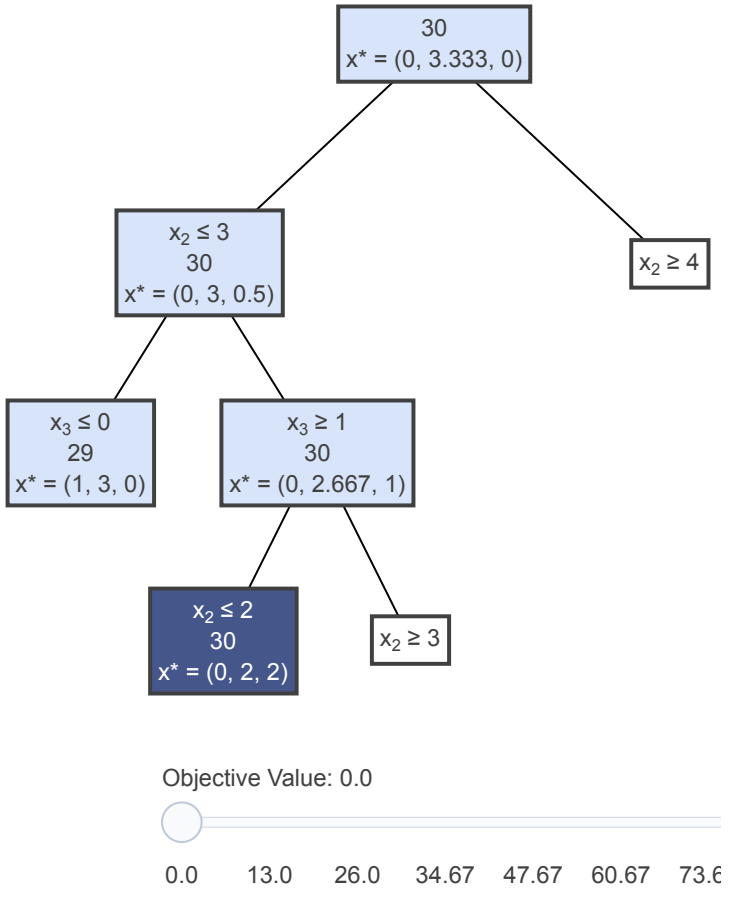
Objective Value: 0.0

0.0 13.0 26.0 34.67 47.67 60.67 73.6

Processing math: 32%

Geometric Interpretation of LPs





Processing math: 32%

In [ ]: