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 [13]: # 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: (Assuming that the problem is a minimization problem). If the optimal solution on a different branch on the same level is feasible integer program solution with an objective value that's the linear program solution's objective value rounded up. (can't do better). Also if the integer solution is found that's better than a different linear program.

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: For maximization, the LP provides an upper bound; for minimization, the LP provides a lower bound.

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: We can fathom the node. Since it's a maximization problem, the LP provides an upper bound of 44.5. The nearest integer rounded down is 44. Since we already have a solution with value 44, we can declare that we can't do better 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: Because if the solution is integer, than we've found a feasible solution to the integer program that is equal to the bound we've placed on the IP by way of the LP solution.

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

A: LP is a relaxation of the IP; if we had a feasible IP solution, then we would actually have a feasible LP solution. However, if we can't have feasible LP, we for sure can't have feasible IP.

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: Minimization 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: It's not fathomed; the only integer solution we found has a value of 20. We know that this is a minimization problem, and we could potentially better with the current z value of 16.3. Need to constrain x_1 for the next two nodes.

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 the children of the current node have a lower bound of 16.3 because their parent is the more relaxed version of the constraints in their program.

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. $$\begin{align*} \max \quad & \sum_{i=1}^n v_ix_i\\ \text{s.t.} \quad & \sum_{i=1}^n w_ix_i \leq W \\ & 0 \leq x_i \leq 1, \text{integer}, i = 1,\dots,n \end{align*}$$

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

```
In [14]: data = pd.read_csv('knapsack_data_1.csv', index_col=0)
         data
```

Out[14]:

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 7 because its weight exceeds W.

```
In [15]: #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 would have been impossible for us to include 7 in the first place.

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 optimal value doesn't change.

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

Out[16]:

	value	weight	value per unit weight
item			
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: While the knapsack isn't full, pick the item with the highest value per unit weight.

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: Items 1, 4; value: 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 [23]:
```
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 o
therwise.
    """
    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 c
apacity
    # recall that we add constraints to the model using m.Add()

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

In [24]:
```
# You do not need to do anything with this cell but make sure you run i
t!
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 [25]:
```
m, x = Knapsack(data, 18)
```

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

In [26]:
```
solve(m)
```

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

Out[26]:
```
{'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: Slightly less; but this is because this is a linear solution

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: Node can't be fathomed because there are potential integer solutions between 64 (heuristic solution) and 70. We should constrain $x_2$ for the next two nodes as $x_2 \leq 0$ and $x_2 \geq 0$

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

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

Out[27]: {'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 [29]: # TODO: Answer Q17
         m, x = Knapsack(data, 18)
         m.Add(x[2] >= 1)
         solve(m)

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

Out[29]: {'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: We can fathom this node because even though it's not an integer solution, it's not better than the feasible integer solution we found earlier.

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 can fathom this node because the best solution it would have after is 64 (because the LP creates an upper bound in this maximization problem), which is a solution we already have.

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 [35]:
```
# 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
```

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

Out[35]:
```
{'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 [36]:
```
# Template
m, x = Knapsack(data, 18)
# Add constraints here
m.Add(x[2] <= 2)
m.Add(x[3] <= 3)
m.Add(x[4] >= 1)
m.Add(x[6] >= 1)
solve(m)
#fathomed? YES
```

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

Out[36]:
```
{'x_1': 0.4, 'x_2': 0.0, 'x_3': 0.0, 'x_4': 1.0, 'x_5': 0.0, 'x_6': 1.
0}
```

A: Final optimal objective value and solution: 64, {'x_1': 1.0, 'x_2': 0.0, 'x_3': 0.0, 'x_4': 1.0, 'x_5': 0.0, 'x_6': 0.0}

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

A: 9 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
$$\begin{align*} \max \quad & 2x_1+x_2\\ \text{s.t.} \quad & x_1 + x_2 \leq 3 \\ & 2x_1 \leq 5 \\ & -x_1 + 2x_2 \leq 2 \\ & x_1, x_2 \geq 0 \end{align*}$$

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

In [37]:
```
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
```