Linear programming

This lab is an introduction to *optimization modeling*. The central idea is that we start from a problem written in words, identify the unknown quantities we want to choose, write an objective that tells us what "best" means, and then write constraints that describe what is allowed.

In this lab, we will see the same modeling style in three different settings:

1. a product-planning example using linear programming,
2. a maximal-flow problem on a directed graph,
3. a shortest-path problem on a weighted directed graph.

Although these problems look different, the common pattern is always:

1. define the variables,
2. define the objective,
3. define the constraints,
4. solve the problem or implement the solution.

Goals

By the end of the lab, you should be comfortable with the following questions:

- How do I translate a text description into variables, objective, and constraints?
- What is the difference between linear programming and integer programming?
- How can a maximal-flow problem be written as a linear program?
- How do I compute a shortest path in a weighted directed graph?

Product planning

Problem statement

A company makes two products A and B. The sales from product A must be at least 80% of all sales $(A + B)$. The market is saturated with 100 products of type A. To produce one product A, the company needs 2 kg of material, and to produce one product B, it needs 4 kg of material. The company has 240 kg of material available. Profit from product A is 20 EUR and profit from product B is 50 EUR. The goal is to maximize total profit.

Solution

Step 1: define the decision variables.

$$A = \text{number of units of product A}, \qquad B = \text{number of units of product B}.$$

Step 2: write the objective function. The company wants as much profit as possible. Since each A gives 20 EUR and each B gives 50 EUR, the objective is:

$$\max(20A + 50B).$$

Step 3: translate the text into constraints. The text gives:

- product A must be at least 80% of all sales:
$$A \geq 0.8(A + B),$$
- at most 100 units of A:
$$A \leq 100,$$
- material limit:
$$2A + 4B \leq 240,$$
- nonnegativity:
$$A \geq 0, \qquad B \geq 0.$$

Step 4: write the model in one place. The full linear program is:

$$\max(20A + 50B)$$

subject to

$$A \geq 0.8(A + B),$$
$$A \leq 100,$$
$$2A + 4B \leq 240,$$
$$A \geq 0, \qquad B \geq 0.$$

Implement the model in Python

After writing the model mathematically, the next step is to implement it in Python. In this lab, the implementation is done with `PuLP`, a Python library for building optimization models and sending them to a solver. In other words, `PuLP` lets us define variables, objectives, and constraints in code in the same order in which we wrote them mathematically.

The following code implements the model above, that is, the original problem in which product A must be at least 80% of all sales:

```python
import pulp as lp

#
# Linear programming - classical introduction
#

model = lp.LpProblem(name="Maximize_Profit", sense=lp.LpMaximize)

A = lp.LpVariable(name="A", lowBound=0)
B = lp.LpVariable(name="B", lowBound=0)

model += (20 * A + 50 * B, "Total_Profit")

model += (A >= 0.8 * (A + B), "Min_Sale_Ratio_A")
model += (A <= 100, "Max_Production_A")
model += (2 * A + 4 * B <= 240, "Material_Limit")

status = model.solve(lp.PULP_CBC_CMD(msg=1))

if model.status == 1:
    print(f"Optimal solution found with total profit: {model.
       objective.value():,.2f} EUR")
    print(f"Units of A to produce: {A.value()}")
    print(f"Units of B to produce: {B.varValue}")
else:
    print("An optimal solution was not found.")
```

If the code is correct, the solver should return the solution

$$A = 80, \qquad B = 20,$$

with total profit

$$20 \cdot 80 + 50 \cdot 20 = 2600.$$

This is a useful first check: the ratio constraint is active, the material constraint is active, and the solution uses the available resources efficiently.

What to notice in the code

- `LpProblem(..., LpMaximize)` creates a maximization problem.
- `LpVariable` creates the decision variables.
- `model += ...` adds either the objective or a constraint.
- `solve(...)` calls the solver.
- `A.value()` and `B.varValue` return the solution.

So the code follows exactly the same order as the mathematical model.

Modify the model to 85%

Now change the condition so that product A must be at least 85% of all sales.

First solve it in exactly the same way as before: keep the model linear and continuous, and only replace 0.8 by 0.85 in the ratio constraint. When you do this, the solver returns a solution of approximately

$$A \approx 88.70, \qquad B \approx 15.65,$$

with profit approximately 2556.52.

This is mathematically a valid linear-programming solution, but it is not an ideal production plan because A and B represent numbers of items, and numbers of items should be whole numbers. That observation motivates the next step: we keep the same objective and constraints, but now require A and B to be integers. This gives an integer-programming model.

Use the following code for that integer version:

```
model = lp.LpProblem(name="Maximize_Profit", sense=lp.LpMaximize)

A = lp.LpVariable(name="A", lowBound=0, cat="Integer")
B = lp.LpVariable(name="B", lowBound=0, cat="Integer")

model += (20 * A + 50 * B, "Total_Profit")

model += (A >= 0.85 * (A + B), "Min_Sale_Ratio_A")
model += (A <= 100, "Max_Production_A")
model += (2 * A + 4 * B <= 240, "Material_Limit")

status = model.solve(lp.PULP_CBC_CMD(msg=0))
```

With the integer restriction and the stronger 85% condition, a correct run should give

$$A = 90, \qquad B = 15,$$

with total profit

$$20 \cdot 90 + 50 \cdot 15 = 2550.$$

Compared with the continuous 85% model, the profit is slightly smaller because the solver is no longer allowed to use fractional values.

At this point, compare the two modeling choices:

- **linear programming**: variables may be fractional,
- **integer programming**: variables must be whole numbers.

Run the continuous 85% version and the integer 85% version, then compare them. In particular, note:

- that strengthening the ratio from 80% to 85% makes the solution more restrictive,
- that the continuous 85% model produces fractional values,

- and that the integer model changes the result slightly further.

For an even more informative comparison, it is useful to distinguish three cases:

1. 80% with continuous variables:
$$A = 80, \qquad B = 20, \qquad \text{profit} = 2600.$$
2. 85% with continuous variables:
$$A \approx 88.70, \qquad B \approx 15.65, \qquad \text{profit} \approx 2556.52.$$
3. 85% with integer variables:
$$A = 90, \qquad B = 15, \qquad \text{profit} = 2550.$$

This comparison shows three important facts: strengthening the ratio from 80% to 85% lowers the best achievable profit, the continuous 85% model allows fractional solutions, and the integer 85% model gives a slightly smaller profit because the solver must choose whole numbers.

Exercise 1: Maximal flow

Problem statement

You have a directed graph $G(V, E)$, where every edge $(u, v) \in E$ has a nonnegative capacity $c(u, v)$. You are also given a source vertex s and a terminal vertex t. Your task is to find the maximal flow from s to t.

Assume the graph consists of vertices

$$V = \{s, t, v_1, v_2, v_3, v_4, v_5\}$$

and capacities:

$$\begin{aligned} &c(s, v_1) = 20, && c(s, v_4) = 7, \\ &c(v_1, v_2) = 15, && c(v_1, v_3) = 5, \\ &c(v_2, t) = 12, && c(v_2, v_5) = 5, \\ &c(v_3, v_4) = 2, && c(v_3, v_5) = 4, \\ &c(v_4, v_1) = 3, && c(v_4, v_5) = 8, \\ &c(v_5, t) = 17. \end{aligned}$$

Solution

Step 1: define one variable per directed edge. For each edge (u, v), define a flow variable f_{uv}. This variable means:

how much flow travels through that edge.

So you should introduce variables such as:

$$f_{s,v_1},\ f_{s,v_4},\ f_{v_1,v_2},\ f_{v_1,v_3}, \ldots$$

Step 2: write the objective. We want as much flow as possible to leave the source s. So the objective should maximize the total outgoing flow from s.

In this graph, s has two outgoing edges, so the objective will involve:

$$f_{s,v_1} + f_{s,v_4}.$$

Step 3: write the capacity constraints. Every edge has a capacity, so for each edge:

$$0 \leq f_{uv} \leq c(u, v).$$

This gives one lower bound and one upper bound for each edge variable.

Step 4: write flow-conservation constraints. At every internal vertex, incoming flow must equal outgoing flow.

So for each of

$$v_1,\ v_2,\ v_3,\ v_4,\ v_5$$

you should write one conservation equation.

For example, at v_2:

whatever enters v_2 must leave v_2.

Step 5: assemble the full LP. Once you have:

- all variables,
- the objective,
- the capacity constraints,
- the conservation constraints,

you have finished the modeling task.

Exercise 2: Shortest path

Problem statement

Given a randomly generated directed graph $G(V, E)$, find the shortest distance from node 1 to node n, where n is the index of the last node.

The given code scaffold

The original Python file contains the following scaffold:

```
import networkx as nx
import matplotlib.pyplot as plt
import random


def make_my_graph(v, e):
    G = nx.DiGraph()
    G.add_node(1, color="green") # Starting node
    G.add_nodes_from(range(2, v + 2), color="red")
    G.add_node(v + 2, color="green") # Ending node

    edges = set()
    while len(edges) < e:
        from_node = random.choice(range(2, v + 2))
        to_node = random.choice(range(2, v + 2))
        if from_node != to_node:
            edges.add((from_node, to_node))

    for (from_node, to_node) in edges:
        G.add_edge(from_node, to_node, cost=random.uniform(0.1, 2))

    G.add_edge(1, random.choice(range(2, v + 1)), cost=1)
    G.add_edge(random.choice(range(2, v + 1)), v + 2, cost=1)

    return G


def plot_graph(G):
    pos = nx.shell_layout(G)
    nx.draw_networkx_nodes(
        G, pos, node_size=200,
        node_color=[data['color'] for v, data in G.nodes(data=True)]
    )
    nx.draw_networkx_edges(G, pos, width=1, alpha=0.5, edge_color='b'
       )
    nx.draw_networkx_labels(G, pos, font_size=10, font_family='sans-
       serif')
    edge_labels = dict([((u, v), round(d['cost'], 3))
                        for u, v, d in G.edges(data=True)])
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
    plt.axis('off')
    plt.show()
```

```
def return_shortest_path(G):
    # List nodes
    print("Vertices:")
    for x in G.nodes():
        print(x)
    # List edges
    print("Edges:")
    for (u, v, d) in G.edges(data=True):
        print(f"Edge from {u} to {v} with cost {d['cost']}.")
    pass  # Implement this function that will return the solution
```

Solution

You are given code that:

- creates a random directed graph,
- assigns a positive cost to each edge,
- plots the graph,
- but leaves the shortest-path function unfinished.

Your job is to complete the function `return_shortest_path(G)`.

Step 1: identify the source and target. In this generated graph:

- the source is node 1,
- the target is the last node, which is the largest node label.

Step 2: understand what the edge weights mean. Every edge has a field called `cost`. The shortest path is the path whose total sum of `cost` values is as small as possible.

Step 3: choose a shortest-path routine. Since all generated costs are positive, you can use a standard shortest-path routine from NetworkX.

The easiest option is to use a function that returns:

- the total distance,
- and the path itself.

Step 4: complete `return_shortest_path(G)`. The function should:

1. print the vertices,
2. print the edges and their costs,
3. compute the shortest path from node 1 to the last node,
4. print the shortest distance,
5. print the path,
6. return the result.

Step 5: run the suggested test.

```
random.seed(12345675)
G = make_my_graph(30, 55)
shortest_path_problem = return_shortest_path(G)
plot_graph(G)
```