CS 5800 Final project report – Solving max-flow problem with linear programming

-Zilei Liu
-Group members: Jieling Gong, Cong Fu

Introduction:

Our group chose linear programming as our project topic due to its applicability in addressing real-world problems that involve maximizing or minimizing objectives subject to constraints. Linear programming's scope extends to a variety of optimization issues where some problems can only be effectively solved using this method. As highlighted in CLRS (Introduction to Algorithms), the generality and power of linear programming make it uniquely suited for these types of problems (Cormen et al., 2009).

We are applying the simplex method to solve linear programming problems. This method, devised by George Dantzig in 1947, is a widely used algorithm for optimizing linear objectives subject to constraints (Dantzig, 1947). To facilitate the process, we use a tableau to keep track of all the coefficients involved.

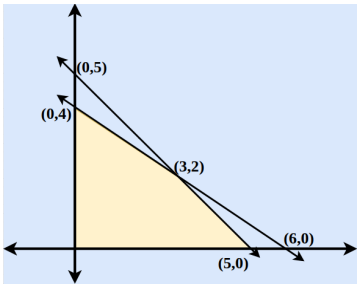
We then apply the general solution of linear programming problems to real-world scenarios. I chose to solve the max-flow problem using linear programming. Let G = (V, E) be a flow network with a capacity function c. The goal of the max-flow problem is to maximize the flow from the source node to the sink node (Cormen et al., 2009).

I chose to solve the max-flow problem because the formulation that transforms a graph problem into equations fascinates me. With the proper formulation, linear programming can solve a variety of problems, including the shortest path, maximum flow, minimum-cost flow, and multicommodity flow. Although there are more efficient algorithms than linear programming for the shortest path and maximum flow problems, linear programming shines in cases like multicommodity flow problems, where no efficient specific algorithms exist to solve them (Cormen et al., 2009). Hence, while other algorithms may be more efficient for solving the max-flow problem, I chose to use linear programming to demonstrate the power of formulation in converting a graph problem into a linear programming one.

Analysis:
1) The general solution of linear programming – using simplex method.
- group work by all group members

First, we introduce the linear programming with an easy example.

Bakery's Problem: We want to maximize the profit from baking cupcakes and cookies with the following constraints:



Profit per cookie: $7
Profit per cupcake: $9
Availability of flour: 12 pounds
Availability of sugar: 5 pounds
Each cookie requires 2 pounds of flour and 1 pound of sugar.
Each cupcake requires 3 pounds of flour and 1 pound of sugar.

The formulation of this problem is quite straight-forward.

Maximize: $7x_1 + 9x_2$
Subject to:
$2x_1 + 3x_2 \leq 12$
$x_1 + x_2 \leq 5$
Where all the variables are non-negative.

Next, we can demonstrate the algorithm of finding the solution using simplex method, the tableau version.

x1	x2	s1	s2	function	value
2	3	1	0	0	12
1	1	0	1	0	5
-7	-9	0	0	1	0

Table (1)

In table (1), there are the two constraints and one objective function to maximize, in standard form. x_1 represents cookies and x_2 represents cupcakes. s_1 and s_2 are slack variables. Notice the leading coefficients (indicators) are negative in a maximization problem. Once there are no negative leading coefficients, we are done and ready to report the optimal answer.

We will find basic variables and the objective function in columns containing only one number, and the rest are zeros. We can read the result directly from the tableau. We use the value in each row to divide the only non-zero number in the same row.

x1	x2	s1	s2	function	value
2	3	1	0	0	12
1	1	0	1	0	5
-7	-9	0	0	1	0

Table (2)

For example, the result in table (2) is s_1=12/1 = 12, s_2=5/1 = 5, (hence x_1 = 0 and x2=0), profit = 0/1 = 0. To interpret this: we will make 0 cookies and 0 cupcakes, getting a profit of 0. This is the starting point of our simplex method.

x1	x2	s1	s2	function	value
2	3	1	0	0	12
1	1	0	1	0	5
-7	-9	0	0	1	0

Table (3)

Then, we need to choose a pivot, and do some Gaussian Eliminations.

We start from the "most negative" indicators (Dantzig's pivot rule). In this problem it is x_2. We then use the value in the last column to divide the value in x_2's column and pick the row with the least answer. In this problem row 1 has 12/3 =4, row2 has 5/1 = 5, so we choose the 3 as the pivot.

In this problem, since cupcakes have more impact on our profit, we try to relax it first. How many cupcakes can we make? According to the two constraints, flour says we can make 12/3 = 4 cupcakes. Sugar says we can make 5/1 = 5 cupcakes. So, considering them, we can make 4 cupcakes.

Now we need to manipulate the rows so we can find basic variables and objective functions in columns where the other values are 0.

We keep row 1 in the table (3) fixed, let row2 = row2 - ⅓* row1, and row3 = row3 + 3 * row1, we will get the desired table (4). Then we can read the result directly. The results in table (4) are: x_2 = 12/3 = 4. s_2 = 1/1 =1. (Hence x_1=0 because we used up the flour) Max profit = 36/1 = 36. We make 4 cupcakes and no cookies and get a profit of 36.

x1	x2	s1	s2	function	value
2	3	1	0	0	12
1/3	0	-1/3	1	0	1
-1	0	3	0	1	36

Table (4)

However, there is still one negative indicator (-1) there. We can still make cookies. We need to pivot again and perform Gaussian Elimination again.

x1	x2	s1	s2	function	value
0	3	3	-6	0	6
1/3	0	-1/3	1	0	1
0	0	2	3	1	39

Table (5)

Table (5) is the final answer, since there are no negative indicators (pivot is ⅓ in x_1's column in Table (4)). Read the results: we can make 6/3 = 2 cupcakes and 1/(⅓) = 3 cookies, hence get a max profit of 39.

To find a general solution for this problem, we use matrix notation to give a standard formulation of a general maximizing linear programming problem:

Maximize: $c^T x$
Subject to: $Ax \leq b$
Where x is a vector of non-negative variables.

In the previous example, $c = \begin{bmatrix} 7 \\ 9 \end{bmatrix}$, $A = \begin{bmatrix} 2 & 3 \\ 1 & 1 \end{bmatrix}$, $b = \begin{bmatrix} 12 \\ 5 \end{bmatrix}$. Then using the algorithm above, we wrote a python code (appendix 1) to solve a general maximizing linear programming problem. The input is c, A and b. The output is the solution x and the optimal value.

Using the example above as a test case.

```
def main():
    # Example usage
    A = [
        [2, 3],
        [1, 1]
        ]
    c = [7, 9]
    b = [12, 5]
    solution, optimal_value = simplex(c, A, b)
    print(f"Optimal solution: x1 = {solution[0]:.1f}, x2 = {solution[1]:.1f}")
    print(f"Optimal value: {optimal_value:.1f}")
```

We can get the expected answer.

```
Optimal solution: x₁ = 3.0, x₂ = 2.0
Optimal value: 39.0
```

Now, we discuss the time and space complexity of this simplex linear programming code.

We let the number of variables be n and the number of constraints be m. The space we need is a 2-D list to keep the tableau. The number of rows is the number of constraints, which is m. And the number of columns is the number of variables. However, notice that we have introduce m slack variables in the simplex method. Hence, the number of columns is (m+n). So, the space complexity is O((m+n)*m).

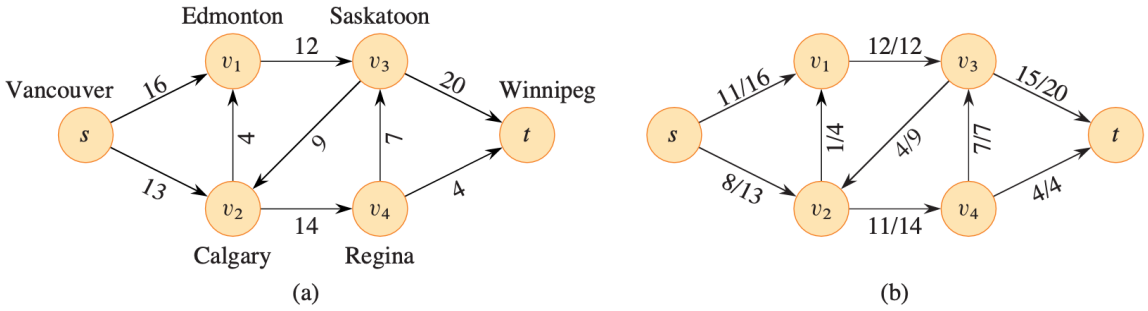
For the time complexity, to initialize the table we need O((m+n)*m). This is clearly not the dominant factor. The major operation is pivoting.

1. Find the pivot column: O(m+n), since we have m+n columns.
2. Find the pivot row: O(m), since we have m rows.
3. Pivot operation: O((m+n)*m), since we have ((m+n)*m entries.

The dominant one is O((m+n) *m). And to get the final solution, we need in the worst case, m times pivoting. Hence, the time complexity is $O((m+n)*m^2)$.

2) The general solution of maximum flow– using linear programming.
- individual work by Zilei Liu

The max flow problem is essential in network optimization, aiming to maximize the flow from a source to a sink in a directed graph where each edge has a nonnegative capacity. A practical application is seen in logistics, such as a trucking network, which transports goods from its Edmonton warehouse (source) to its Winnipeg warehouse (sink) through cities like Saskatoon, Regina, and Calgary. This example is from CLRS (Introduction to Algorithms). Then we can model the cities as nodes. The source node is s. The sink node is t. And the internodes are v_1, v_2, v_3 and v_4 (Cormen et al., 2009).



Our goal is to find the flow function f maximize the flow from source node to sink node.

	s	v_1	v_2	v_3	v_4	t
s	0	16	13	0	0	0
v_1	0	0	0	12	0	0
v_2	0	4	0	0	14	0
v_3	0	0	9	0	0	20
v_4	0	0	0	7	0	4
t= 5	0	0	0	0	0	0

	s	v_1	v_2	v_3	v_4	t
s	0	f_{sv_1}	f_{sv_2}	f_{sv_3}	f_{sv_4}	f_{st}
v_1	f_{v_1s}	0	$f_{v_1v_2}$	$f_{v_1v_3}$	$f_{v_1v_4}$	f_{v_1t}
v_2	f_{v_2s}	$f_{v_2v_1}$	0	$f_{v_2v_3}$	$f_{v_2v_4}$	f_{v_2t}
v_3	f_{v_3s}	$f_{v_3v_1}$	$f_{v_3v_2}$	0	$f_{v_3v_4}$	f_{v_3t}
v_4	f_{v_4s}	$f_{v_4v_1}$	$f_{v_4v_2}$	$f_{v_4v_3}$	0	f_{v_4t}
t	f_{ts}	f_{tv_1}	f_{tv_2}	f_{tv_3}	f_{tv_4}	0

The capacity and flow tables

In a directed graph, each edge has a nonnegative capacity, representing the maximum possible flow along that edge. To represent the capacity function c efficiently, we can use a table. When there is no edge between two nodes, the capacity of the edge is 0. The same applies to self-edges, which also have a capacity of 0. The goal is to find the corresponding flows for each edge, represented in a similar flow table.

To solve this in linear programming, we need the following formulation (Cormen et al., 2009).

$$
\begin{aligned}
\text{maximize} \quad & \sum_{v \in V} f_{sv} - \sum_{v \in V} f_{vs} \\
\text{subject to} \quad & \\
& f_{uv} \leq c(u,v) \quad \text{for each } u, v \in V \\
& \sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv} \quad \text{for each } u \in V - \{s,t\} \\
& f_{uv} \geq 0 \qquad \text{for each } u, v \in V \ .
\end{aligned}
$$

The objective of the max flow problem is to maximize the sum of the flows that leave the source node. To achieve this, we need to consider two sets of constraints:

1. **Capacity Constraints**: The flow along each edge cannot exceed its corresponding capacity. This ensures that the flow through any edge does not surpass the maximum limit specified for that edge.
2. **Conservation Constraints**: For every intermediate node (i.e., nodes other than the source and sink), the sum of the flows coming into the node must equal the sum of the flows going out of the node. This constraint ensures that flow is conserved at each node.

With our example, the formulation is as following.

Maximize: $f_{sv_1} + f_{sv_2} + f_{sv_3} + f_{sv_4}$

Subject to:

$f_{sv_1} \leq 16$
$f_{sv_2} \leq 13$
$f_{v_1v_3} \leq 12$
$f_{v_2v_1} \leq 4$
$f_{v_2v_4} \leq 14$
$f_{v_3v_2} \leq 9$
$f_{v_3t} \leq 20$
$f_{v_4v_3} \leq 7$
$f_{v_4t} \leq 4$
$f_{sv_1} + f_{v_2v_1} + f_{v_3v_1} + f_{v_4v_1} + f_{tv_1} = f_{v_1s} + f_{v_1v_2} + f_{v_1v_3} + f_{v_1v_4} + f_{v_1t}$
$f_{sv_2} + f_{v_1v_2} + f_{v_3v_2} + f_{v_4v_2} + f_{tv_2} = f_{v_2s} + f_{v_2v_1} + f_{v_2v_3} + f_{v_2v_4} + f_{v_2t}$
$f_{sv_3} + f_{v_1v_3} + f_{v_2v_3} + f_{v_4v_3} + f_{tv_3} = f_{v_3s} + f_{v_3v_1} + f_{v_3v_2} + f_{v_3v_4} + f_{v_3t}$
$f_{sv_4} + f_{v_1v_4} + f_{v_2v_4} + f_{v_3v_4} + f_{tv_4} = f_{v_4s} + f_{v_4v_1} + f_{v_4v_2} + f_{v_4v_3} + f_{v_4t}$

To solve the max flow problem using our general linear programming solver, we need to convert the flow network formulation into standard matrix notation. Specifically, we need to transform the capacity matrix of the flow network into the standard inputs c, A, and b for the linear programming problem.

c: The vector of coefficients for the objective function.
A: The matrix of constraints.
b: The vector representing the bounds for the constraints.

To achieve this, I have written a Python code (Appendix 2) that performs this conversion.

```
def main():
    # Example usage
    net_flow = [
        [0, 16, 13, 0, 0, 0],
        [0, 0, 0, 12, 0, 0],
        [0, 4, 0, 0, 14, 0],
        [0, 0, 9, 0, 0, 20],
        [0, 0, 0, 7, 0, 4],
        [0, 0, 0, 0, 0, 0],
        ]
    c, A, b = create_lp_matrices(net_flow)
    print("Objective function coefficients (c):")
    print(c)

    print("\nConstraints matrix (A):")
    for i, row in enumerate(A):
        print(f"Row {i + 1}: {row}")

    print("\nRight-hand side vector (b):")
    print(b)
```

```
Objective function coefficients (c):
[0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Constraints matrix (A):
Row 1: [0, -1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0]
Row 2: [0, 1, 0, 0, 0, 0, -1, 0, -1, -1, -1, -1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
Row 3: [0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0]
Row 4: [0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, -1, -1, 0, -1, -1, -1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0]
Row 5: [0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0]
Row 6: [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, -1, -1, -1, 0, -1, -1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0]
Row 7: [0, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, -1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, -1, 0]
Row 8: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, -1, -1, -1, -1, 0, -1, 0, 0, 0, 0, 1, 0]
Row 9: [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 10: [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 11: [0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
Row 12: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 13: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 14: [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 15: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 16: [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 17: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 18: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 19: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 20: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 21: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 22: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 23: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 24: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 25: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 26: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 27: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 28: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 29: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 30: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Row 31: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
Row 32: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
Row 33: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
Row 34: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
Row 35: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
Row 36: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
Row 37: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
Row 38: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0]

Right-hand side vector (b):
[0, 0, 0, 0, 0, 0, 0, 0, 16, 13, 0, 0, 0, 0, 0, 12, 0, 0, 0, 4, 0, 14, 0, 0, 0, 9, 0, 20, 0, 0, 0, 7, 4, 0, 0, 0, 0, 0]
```

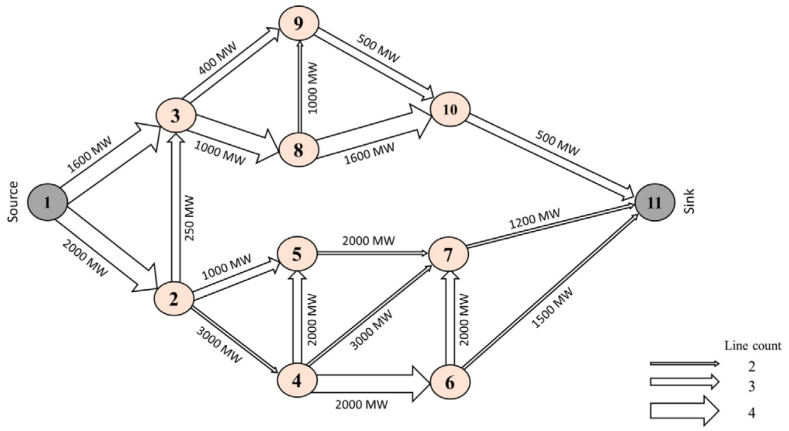
Finally, feed this as the input to the linear programming solver (Appendix 3). We can find the solution of the maximum flow problem.

```
Solution:
        0      1      2      3      4      5
 0 [     0   12.0   11.0    0.0    0.0    0.0]
 1 [   0.0      0      0   12.0    0.0    0.0]
 2 [   0.0      0      0    0.0   11.0    0.0]
 3 [   0.0      0      0      0      0   19.0]
 4 [   0.0      0      0    7.0      0    4.0]
 5 [   0.0      0      0      0      0      0]

Objective Value: 23.0
```

This solution is correct. And now given any capacity function of a maximum flow problem, we can find the solution using our program.

Here is another example Maximum flow in electrical network (Ahmadi & Salehi, 2021).



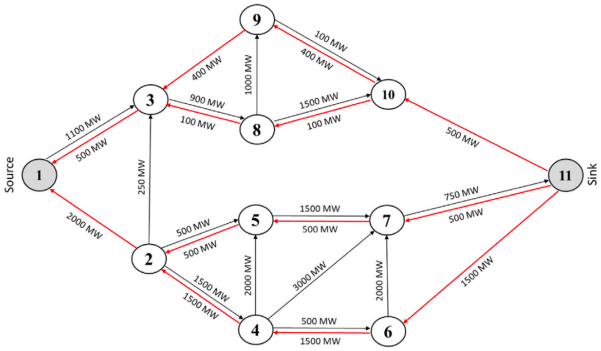
The matrix of the capacity function is as following:

	1	2	3	4	5	6	7	8	9	10	11
1	0	2000	1600	0	0	0	0	0	0	0	0
2	0	0	250	3000	1000	0	0	0	0	0	0
3	0	0	0	0	0	0	0	1000	400	0	0
4	0	0	0	0	2000	2000	3000	0	0	0	0
5	0	0	0	0	0	0	2000	0	0	0	0
6	0	0	0	0	0	0	2000	0	0	0	1500
7	0	0	0	0	0	0	0	0	0	0	1200
8	0	0	0	0	0	0	0	0	1000	1600	0
9	0	0	0	0	0	0	0	0	0	500	0
10	0	0	0	0	0	0	0	0	0	0	500
11	0	0	0	0	0	0	0	0	0	0	0

Feed this to the program as input (The same code as appendix 3 with the above matrix as input), we will have the correct solution.

```
Solution:
       1      2      3      4      5      6      7      8      9     10     11
 1 [     0 2000.0  500.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0    0.0]
 2 [   0.0      0      0 2000.0      0    0.0    0.0      0      0      0    0.0]
 3 [   0.0    0.0      0    0.0    0.0    0.0    0.0  500.0    0.0    0.0    0.0]
 4 [   0.0      0      0      0      0 1500.0  500.0      0      0      0    0.0]
 5 [   0.0    0.0      0    0.0      0    0.0    0.0      0      0      0    0.0]
 6 [   0.0      0      0      0      0      0      0      0      0      0 1500.0]
 7 [   0.0      0      0      0      0      0      0      0      0      0  500.0]
 8 [   0.0    0.0      0    0.0    0.0    0.0    0.0      0      0  500.0    0.0]
 9 [   0.0    0.0      0    0.0    0.0    0.0    0.0      0      0      0    0.0]
10 [   0.0    0.0      0    0.0    0.0    0.0    0.0      0      0      0  500.0]
11 [   0.0      0      0      0      0      0      0      0      0      0      0]

Objective Value: 2500.0
```



solution

The same solution as found by Ahmadi & Salehi (2021).

Finally, let us discuss the space and time complexity again. Given a matrix a max flow capacity function, we need to use the code (Appendix 2) to convert it into the standard input of the linear programming solver.

Let the number of vertices of the network flow be n. It is easy to see the dominant factor of space complexity is the storage used to for the constraint matrix. The number of columns of the matrix is the number of variables we have. We need a matrix of flows, that is $|V|^2$ variables. And the number of rows is about $|V|^2$ for capacity constraints and 2(|V|-2) conservation constraints (This is because for equality constraint, we used 2 inequities to represent 1 constraint). The dominant factor is $|V|^2$. Hence, for the whole matrix, the space complexity is $O(|V|^4)$, where |V| is the number of vertices. And the time complexity is also $O(|V|^4)$ because we simply fill out the matrix without many complex calculations.

Recall that the space complexity for the linear programming solver is O((m+n)*m) and the time complexity is $O((m+n)*m^2)$, where the m is number of constraints and n is the number of variables.

So, now the number of variables is $|V|^2$ and the number of constraints is also $|V|^2$. If we combine these 2 programs, the space complexity is $O(|V|^4)$ and the time complexity is $O(|V|^6)$.

Conclusion:

With the linear programming and simplex method, we can solve the max-flow problem accurately just like any other algorithms. As is mentioned earlier, the purpose of this project is to find out the power and the generality of the formulation of linear program. There are 2 major limitations about this project.

1. Solving max flow with linear programming is not efficient enough. The time complexity as mentioned earlier is $O(|V|^6)$. Most specific algorithm aiming to solve max flow would do much better. For example, the Edmonds-Karp method has a time complexity of $O(|V| * |E|^2)$. In the case the graph is extremely dense when $|E|$ is close to $|V|^2$, the Edmonds-Karp method is still better. In most cases where $|E|$ is close to $|V|$, the Edmonds-Karp method is about $O(|V|^3)$, a lot better than $O(|V|^6)$.

2. In the implementation of Python code, for convenience, we often create $|V|^2$ variables. This approach leads to a significant waste of time and space. When there is no edge between two nodes, we do not need to include this flow in the matrix. Therefore, there is room for optimization.

From this project, I have gained a lot from the formulation of linear programming. Theoretically, it can be done, but the real implementation is still fascinating. Learning how to automate the process of converting the capacity matrix into a constraints matrix with code has been a lot of fun. I also spent considerable time debugging the code. One major issue I solved was using two inequalities to represent one equality in the conservation constraint.

$$f_{sv_1} + f_{v_2v_1} + f_{v_3v_1} + f_{v_4v_1} + f_{tv_1} - f_{v_1s} - f_{v_1v_2} - f_{v_1v_3} - f_{v_1v_4} - f_{v_1t} = 0$$

To use this constraint, I substituted it with two inequalities to make it compatible with the linear programming solver:

$$f_{sv_1} + f_{v_2v_1} + f_{v_3v_1} + f_{v_4v_1} + f_{tv_1} - f_{v_1s} - f_{v_1v_2} - f_{v_1v_3} - f_{v_1v_4} - f_{v_1t} \leq 0$$
$$-f_{sv_1} - f_{v_2v_1} - f_{v_3v_1} - f_{v_4v_1} - f_{tv_1} + f_{v_1s} + f_{v_1v_2} + f_{v_1v_3} + f_{v_1v_4} + f_{v_1t} \leq 0$$

This project is valuable to me for many reasons. I now know how to implement a linear programming solver by myself and how to use existing ones. Additionally, I understand how optimization problems can be generally formulated into linear programming problems. This is a very powerful tool in my programming toolkit, and it will be immensely beneficial in my future learning and projects.

Appendix 1 (linear programming solver with simplex method tableau version)

```
def simplex(c, A, b):

    num_vars = len(c)
    num_constraints = len(b)
    tableau = []
    # Step 1:
    for coefficients, rhs_value in zip(A, b):
        slack_variables = [0] * num_constraints
        tableau_row = coefficients + slack_variables + [rhs_value]
        tableau.append(tableau_row)
    # Step 2:
    negated_objective_coefficients = []
    for coeff in c:
        negated_coeff = -coeff
        negated_objective_coefficients.append(negated_coeff)
    objective_function_row = negated_objective_coefficients + [0] * (num_constraints +
1)
    tableau.append(objective_function_row)

    # Adding the identity matrix to form slack variables
    for i in range(num_constraints):
        tableau[i][num_vars + i] = 1

    def pivot(pivot_row_index, pivot_col_index):
        pivot_element = tableau[pivot_row_index][pivot_col_index]
        if pivot_element == 0:
            raise ValueError("Pivot element is zero, which is not allowed.")

        # Step 1: Normalize the pivot row
        normalized_pivot_row = []
        for element in tableau[pivot_row_index]:
            normalized_element = element / pivot_element
            normalized_pivot_row.append(normalized_element)
        tableau[pivot_row_index] = normalized_pivot_row

        # Step 2: Eliminate the pivot column in all other rows
        for row_index in range(len(tableau)):
            if row_index != pivot_row_index:
                pivot_col_value = tableau[row_index][pivot_col_index]
                updated_row = []
                for col_index, current_row_element in enumerate(tableau[row_index]):
                    updated_element = current_row_element - (pivot_col_value *
normalized_pivot_row[col_index])
                    updated_row.append(updated_element)
                tableau[row_index] = updated_row
```

```
# Simplex iterations
while True:
    # Find the pivot column (most negative value in the last row)
    num_decision_vars = len(tableau[0]) - 1
    pivot_col = 0
    for col_index in range(1, num_decision_vars):
        current_coefficient = tableau[-1][col_index]
        if current_coefficient < tableau[-1][pivot_col]:
            pivot_col = col_index

    # Optimal reached checking
    if tableau[-1][pivot_col] >= 0:
        break

    # Find the pivot row
    ratios = []
    for r in range(num_constraints):
        if tableau[r][pivot_col] > 0:
            ratio = tableau[r][-1] / tableau[r][pivot_col]
            ratios.append((ratio, r))
    if not ratios:
        raise ValueError("The problem is unbounded.")

    # Find the row with the smallest ratio, which is the pivot row
    pivot_row = min(ratios)[1]

    # Perform the pivot operation
    pivot(pivot_row, pivot_col)

# Extract solution
solution = [0] * num_vars
for i in range(num_vars):
    col = [row[i] for row in tableau[:-1]]
    if col.count(1) == 1 and col.count(0) == num_constraints - 1:
        solution[i] = tableau[col.index(1)][-1]

optimal_value = tableau[-1][-1]
return solution, optimal_value
```

Appendix 2 (The code to convert matrices)

```
def create_lp_matrices(net_flow, source=0, sink=None):
    n = len(net_flow)  # Number of nodes
    if sink is None:
        sink = n - 1  # Default sink is the last node

    # Objective function coefficients (c) for maximizing flow from source to sink
    c = [0] * (n * n)
    for j in range(n):
        if source != j:
            c[source * n + j] = 1

    # Constraints matrix (A)
    A = []
    b = []

    # Flow conservation constraints (for each node except for source and sink)
    for i in range(1, n-1):
        row = [0] * (n * n)
        # Outflow from node i
        for j in range(n):
            if i != j:
                row[i * n + j] = 1
        # Inflow to node i
        for j in range(n):
            if i != j:
                row[j * n + i] = -1
        A.append(row)
        b.append(0)

        # add the same constraints in the opposite way
        row_opposite = [-x for x in row]
        A.append(row_opposite)
        b.append(0)

    # Capacity constraints for each edge
    for i in range(n):
        for j in range(n):
            if i != j:
                row = [0] * (n * n)
                row[i * n + j] = 1
                A.append(row)
                b.append(net_flow[i][j])

    return c, A, b
```

Appendix 3 (Code to solve the max-flow example)

```
from create_matrices import create_lp_matrices
from LP import simplex

def list_to_matrix(flat_list):
    import math

    n = int(math.sqrt(len(flat_list)))  # Determine the size of the matrix
    matrix = []

    for i in range(n):
        row = flat_list[i*n:(i+1)*n]
        matrix.append(row)

    return matrix

def print_matrix(matrix):
    n = len(matrix)
    print("   " + " ".join(f"{j:>6}" for j in range(n)))
    for i, row in enumerate(matrix):
        formatted_row = " ".join(f"{value:>6.1f}" if isinstance(value, float) else
f"{value:>6}" for value in row)
        print(f"{i:>2} [{formatted_row}]")

def main():
    # Example usage
    net_flow = [
        [0, 16, 13, 0, 0, 0],
        [0, 0, 0, 12, 0, 0],
        [0, 4, 0, 0, 14, 0],
        [0, 0, 9, 0, 0, 20],
        [0, 0, 0, 7, 0, 4],
        [0, 0, 0, 0, 0, 0],
        ]
    c, A, b = create_lp_matrices(net_flow)
    solution, objective_value = simplex(c, A, b)

    # Print the results
    print("Solution:")
    solution_matrix = list_to_matrix(solution)
    print_matrix(solution_matrix)
    print(f"\nObjective Value: {objective_value}")
```

References

Bulut, M., & Özcan, E. (2021). Optimization of electricity transmission by Ford–Fulkerson algorithm. *Sustainable Energy Grids and Networks*, *28*, 100544. https://doi.org/10.1016/j.segan.2021.100544

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms, fourth edition*. MIT Press.

Iyer, V. (2022, January 4). Simplex Method : The Easy Way - Vijayasri Iyer - Medium. *Medium*. https://vijayasriiyer.medium.com/simplex-method-the-easy-way-f19e61095ac7

Wikipedia contributors. (2024, July 5). *Simplex algorithm*. Wikipedia. https://en.wikipedia.org/wiki/Simplex_algorithm