Amirkabir University of Technology
(Tehran Polytechnic)

Using the Column Generation Algorithm to Solve the Linear Programming Problem

Mohsen Jafari[a*], Mahmoud Mesbah Namini[a], Reza Shafaii Amlashi[a]

[a] *Department of Civil & Environment, AmirKabir University of Technology, Tehran, Iran.*

Abstract

This research explores the application of the column generation algorithm in optimizing transportation networks and addressing the cutting stock problem. Supply and demand dynamics form the backbone of transportation network systems and are crucial for efficient travel flow. Because of its significance, managing supply and demand in complex transportation networks poses challenges, necessitating innovative optimization solutions. The column generation algorithm, an optimization technique used to solve large-scale linear programming problems and renowned for its efficiency in solving large-size optimization problems, is investigated within the context of transportation network optimization and the cutting stock problem. While the algorithm successfully solves transportation network problems, its convergence to the optimal solution is slower and less efficient compared to built-in solvers. However, it excels in applications like the cutting stock problem, where it minimizes waste and optimizes material usage effectively. The column generation algorithm solved the cutting stock problem in a single iteration, while the conventional simplex method solved it in four iterations, indicating the high computational power of the column generation algorithm and its rapid rate of reaching the optimal solution. Through comprehensive problem formulations, algorithmic implementations, and solution validations, this research sheds light on the algorithm's versatility and practical implications in diverse optimization domains.

Keywords: Column Generation Algorithm, Linear Programming, Transportation Networks, Cutting Stock Problem, Optimization.

1. Introduction

Supply and demand dynamics form the backbone of network transportation systems, serving as the lifeblood that keeps goods flowing efficiently from point A to point B [1]. In the intricate web of global logistics, managing supply and demand becomes increasingly complex, necessitating innovative solutions to optimize routes, minimize costs, and meet consumer expectations. At the heart of this challenge lies the network optimization problem, where the goal is to find the most efficient routes and schedules for transporting goods or services from multiple origins to multiple destinations, all while considering various constraints and objectives. The importance of this

* Corresponding author. Tel.: +98-903-729-0601; e-mail: m.jafari.kwf@aut.ac.ir.

problem cannot be overstated, as it directly affects the bottom line of businesses and the quality of service provided to consumers.

In addition to optimizing goods and service transport, efficient human movement is an equally critical component in modern society. From daily commutes to long-distance travel for business or leisure, transportation networks play a pivotal role in facilitating societal functioning and economic growth. Efficient transportation systems not only alleviate traffic congestion and reduce environmental pollution but also enhance accessibility, allowing individuals to access employment opportunities, education, healthcare, and various amenities [2][3]. Minimizing network costs through optimized transportation routes and modes further supports economic efficiency and sustainability efforts, ensuring that resources are allocated effectively to meet the diverse needs of communities [4]. Recognizing the profound impact of transportation on both individual lives and societal development underscores the importance of continually improving and optimizing transportation networks to meet the evolving needs of a dynamic and interconnected world.

The column generation algorithm has gained significant attention for tackling complex problems. George Dantzig and Philip Wolfe originally developed this algorithm in the 1960s for linear programming problems [5], and since then, they have adapted and refined it to address a wide array of optimization challenges, including network optimization. The column generation algorithm operates by iteratively generating and adding columns (variables) to the problem formulation, focusing on the most promising ones to gradually improve the solution [6]. This efficient exploration of the solution space often leads to substantial reductions in computational time and resources compared to traditional methods. However, certain scenarios, particularly those dealing with the inherent complexities of small-scale transportation networks, may limit the effectiveness of the column generation algorithm [7]. This research demonstrates the effectiveness of the column generation algorithm in optimizing travel costs within a small-scale transportation network, highlighting its capability to efficiently solve complex transportation problems and revealing insights into the structure and constraints of such systems.

Lastly, the focus has been shifted to a classic optimization problem, the cutting stock problem. By leveraging the capabilities of the column generation algorithm, its potential for efficiently solving this quintessential problem has been demonstrated, underscoring the algorithm's versatility and adaptability across diverse optimization domains [7].

The paper begins with a literature review and practical applications of the column generation algorithm within optimization contexts, focusing particularly on its relevance to transportation networks. Subsequently, the methodology section elaborates on the column generation algorithm. The next section formally states the main problem and the complementary problem. The results and discussion section presents the solutions to the problems, followed by a conclusion in the final section.

2. Literature review

The column generation algorithm is an iterative optimization technique that is particularly useful for solving large-scale, linear optimization problems with a very large number of variables (or columns). The core idea of the algorithm is to start with a restricted master problem that contains only a small subset of the variables and then iteratively generate new variables (or columns) that have the potential to improve the objective function. The column generation algorithm is widely used in transportation networks to optimize various aspects such as vehicle routing, scheduling, and network design.

Many types of VRPs, including the capacitated VRP, the VRP with time windows, and the VRP with pickup and delivery, have seen widespread application of the column generation algorithm. By formulating the VRP as a set partitioning problem, the column generation approach can efficiently generate routes for vehicles while considering constraints such as capacity, time windows, and customer locations [8].

In transportation networks, such as airlines and railways, the crew scheduling and rostering problem is critical for efficient operations. The column generation algorithm has been used to optimize crew schedules and rosters while considering factors like work rules,

preferences, and cost minimization. Barnhart et al.'s analysis examines the application of column generation techniques for solving network routing problems across diverse domains, including vehicle routing and airline crew scheduling. Through a discussion of various formulations and solution methods, it underscores the pivotal role of column generation in enhancing route planning and resource utilization within transportation networks [9].

The column generation algorithm has been applied to optimize the design and planning of transportation networks, such as determining the optimal locations of hubs, the capacity of transportation links, and the routing of goods or passengers [10].

The column generation algorithm has been used to optimize intermodal transportation systems, where goods or passengers are transported using a combination of different modes (e.g., truck, rail, ship). This involves coordinating the schedules and routes of different modes to minimize cost and maximize efficiency [11].

Meloet et al. proposed a column generation approach that considers constraints such as vehicle capacity, time windows, and customer preferences. By demonstrating the effectiveness of column generation in addressing complex urban logistics challenges, the paper offers practical solutions to optimize delivery routes, minimize costs, and improve service quality [12].

Lastly, Desaulniers et al. introduced a column generation approach that factors in passenger demand, vehicle scheduling, and route design. Through empirical demonstrations, the paper showcases how column generation can elevate the efficiency and quality of public transportation systems, providing transit agencies and urban planners with invaluable tools to enhance accessibility and sustainability in urban areas [13].

These references provide a solid foundation for understanding the column generation algorithm and its practical applications in transportation network optimization (Table 1). The key advantage of the column generation algorithm in transportation network optimization is its ability to efficiently handle large-scale, complex problems with a large number of variables. By iteratively generating new columns (e.g., routes, schedules, and network designs) and updating the restricted master problem, the algorithm can find optimal or near-optimal solutions while considering various operational constraints and objectives.

Table 1. Summary of literature review.

Reference	Topic	Application
[8]	Vehicle Routing	Efficiently generates routes considering constraints.
[9]	Crew Scheduling	Optimizes crew schedules and rosters.
[10]	Transportation Network	Determines optimal network design.
[11]	Intermodal Transportation	Minimizes cost and maximizes efficiency.
[12]	Urban Logistics	Optimizes urban delivery routes.
[13]	Public Transportation	Enhances transportation system efficiency and quality

3. Methodology

The column generation method is a powerful technique employed in mathematical optimization for solving large-scale combinatorial optimization problems. The method involves iteratively generating and adding columns (variables) to the problem formulation to improve the solution's objective function. Here's an outline of the methodology [14]:

- Step 1. Formulate the original problem as a master problem (or main problem), which includes the decision variables (columns) that form an initial feasible solution. The master problem is the original optimization problem, but with a restricted set of variables (columns) that represent an initial feasible solution. This initial set of

variables may not be optimal, but it provides a starting point for the optimization process. The master problem is typically a linear or mixed-integer linear program.

- Step 2. Based on the master problem, define a subproblem (or pricing problem) that searches for new variables (columns) that have the potential to improve the master problem's objective function. The subproblem is a separate optimization problem that is used to identify new variables (columns) that could improve the objective function of the master problem. The subproblem typically has a different objective function and/or constraints compared to the master problem. The subproblem is designed to find variables (columns) that are not yet part of the master problem but could potentially improve its objective function.

- Step 3. The subproblem tries to identify variables (columns) that are not yet part of the master problem but could help reduce the objective function value (in a minimization problem). The subproblem solves a smaller, more focused optimization problem to find new variables (columns) that have the potential to improve the objective function of the master problem. This is typically done by exploiting the structure of the problem and using specialized algorithms such as branch and bound.

- Step 4: If the subproblem discovers a new variable (column) that enhances the objective, it incorporates this variable into the master problem, resuming the process from step 2. The master problem is then re-solved with the updated set of variables (columns), and the process of solving the subproblem and updating the master problem is repeated.

- Step 5: The iterations persist until the subproblem can no longer identify any improving variables (columns), at which point the current solution to the master problem is considered optimal.

The key idea is to start with a restricted master problem and iteratively expand it by solving a subproblem to identify promising new variables to include. This can be more efficient than trying to solve the full problem directly, especially for large-scale optimization problems. Figure 1 visually depicts the above-described procedure and highlights the iterative nature of the column generation approach.

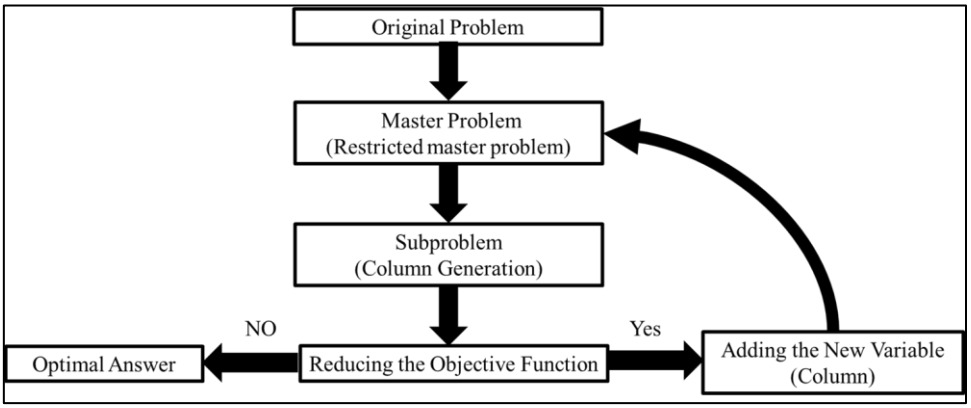

Figure 1. Column generation algorithm process.

4. Problems Statement

This research presents two problems: the first focuses on optimizing the travel cost between supply and demand in a transportation network, while the second further elaborates on the classic cutting stock problem. The next two subsections outline the two issues.

4.1. Transportation problem

The problem is a classic transportation problem involving two origins, denoted as a and b, and three destinations, labeled as 1, 2, and 3 (Figure 2). Table 2 provides the relevant information, including the cost of transporting one unit from origin i to destination j, the maximum capacity of the origins, and the required demand at the destinations. The objective is to determine the optimal transportation plan that satisfies the demand at the destinations while minimizing the total transportation cost [15].

Table 2. Parameters of the transportation problem.

Destination / Origin	1	2	3	Capacity
A	8	6	3	70
B	2	4	9	40
Demand	40	35	25	

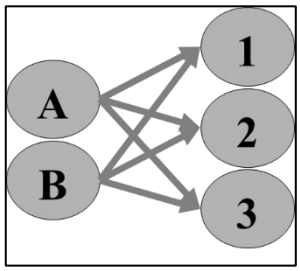

Figure 2. Transportation network shape.

Equations 1 through 8 form the standard mathematical formulation of the problem. Equation 1 represents the objective function; equations 2 and 3 are the supply constraints; equations 4, 5, and 6 are the demand constraints; and equations 7 and 8 refer to the integer and non-negativity constraints. There are 6 decision variables, x_{ij}, where i represents the origin (a or b) and j represents the destination (1, 2, or 3). The problem also has 2 slack variables, 3 excess variables, and 3 artificial variables. The value of M in the objective function can be a large number in the range of 10^3.

$$Minimize$$
$$Z = 8x_{A1} + 6x_{A2} + 3x_{A3} + 2x_{B1} + 4x_{B2} + 9x_{B3} + Ma_1 + Ma_2 + Ma_3 \quad (1)$$

$$S.T.$$
$$x_{A1} + x_{A2} + x_{A3} + s_1 = 70 \quad (2)$$
$$x_{B4} + x_{B5} + x_{B6} + s_2 = 40 \quad (3)$$
$$x_{A1} + x_{B4} - e_1 + a_1 = 40 \quad (4)$$
$$x_{A2} + x_{B5} - e_2 + a_2 = 35 \quad (5)$$
$$x_{A3} + x_{B6} - e_3 + a_3 = 25 \quad (6)$$
$$x_{A1}, x_{A2}, x_{A3}, x_{B4}, x_{B5}, x_{B6} \geq 0 \text{ and integer} \quad (7)$$
$$s_1, s_2, e_1, e_2, e_3, a_1, a_2, a_3 \geq 0 \text{ and integer} \quad (8)$$

4.2. Cutting stock problem

The cutting stock problem is a classic example. In this case, there are stocks 15 meters in length available. The requirement is for 80 pieces of 4-meter stocks, 50 pieces of 6-meter stocks, and 100 pieces of 7-meter stocks. The goal is to reduce the amount of waste [14].

Equations 9 through 13 form the standard mathematical formulation of the problem. Equation 9 represents the objective function; equations 10, 11, and 12 are the demand constraints; and equation 13 refers to the integer and non-negativity constraints. There are 6 decision variables, denoted by x_i, where i represents the pattern of cutting stocks available in Table 3, based on the lengths of demands and the total length of stock.

$$Minimize$$
$$Z = x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \quad (9)$$
$$S.t.$$
$$3x_1 + 2x_4 + 2x_5 = 80 \quad (10)$$
$$2x_2 + x_5 + x_6 = 50 \quad (11)$$
$$x_3 + x_4 + x_6 = 100 \quad (12)$$
$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \text{ and integer} \quad (13)$$

Table 3. variables of cutting stock problem.

	Patterns					
	1	2	3	4	5	6
a_4	3	0	0	2	2	0
a_6	0	2	0	0	1	1
a_7	0	0	2	1	0	1
Waste	3	3	1	0	1	2

5. Result and discussion

This section aims to solve the problems and present the results, drawing on Section 3's explanations regarding the column production algorithm and Section 4's problems.

5.1. Transportation problem

Equations 1 to 8 formulate the transportation problem in the standard mathematical form, as stated in Section 4.1. Considering the explanations provided in Section 3 about the column generation algorithm, and for further information, one can refer to the book "Operations Research: Applications and Algorithms" [14], the master problem (the restricted master problem) is presented in equations 14 to 17, which is a basic solution for the original problem.

$$Minimize\ Z = \sum_{k\in K} \lambda_k \times \left(\sum_{i\in I} \sum_{j\in J} c_{ij} \times P_{ij}^k \right) \quad (14)$$

$$S.t.$$

$$\sum_{k\in K} \lambda_k \times \left(\sum_{i\in I} P_{ij}^k \right) \geq d_j \quad \forall j \in J \quad (15)$$

$$\sum_{k\in K} \lambda_k \times \left(\sum_{j\in J} P_{ij}^k \right) \leq s_i \quad \forall i \in I \quad (16)$$

$$\lambda_k \geq 0 \quad \forall k \in K \quad (17)$$

Let K be the set of current patterns (also called columns). The indices i, j, and k represent elements from the sets of origins, destinations, and patterns, respectively. The parameters used in the model include: c_{ij}, which is the cost of transportation from origin i to destination j; d_j, which represents the demand at destination j; s_i, which is the capacity at origin i; and P_{ij}^k, which is the amount shipped from origin i to destination j in pattern k. The decision variable is λ_k, which indicates how many times pattern k is used in the solution, and it must be greater than or equal to zero. Patterns are used to represent feasible shipment plans that satisfy supply and demand constraints, allowing the algorithm to efficiently explore and combine different options during the column generation process.

The subproblem in the column generation framework is responsible for generating a new pattern (or column) that has a negative reduced cost, which, if added to the master problem, can improve the overall solution. The parameters used in the subproblem include: π_j, which is the dual price associated with the demand constraint at destination j, and μ_i, which is the dual price associated with the capacity constraint at origin i. The decision variable is x_{ij}, representing the amount shipped from origin i to destination j in the new pattern. This variable must be non-negative. The objective of the subproblem is to minimize the reduced cost of a potential new column, defined as the total adjusted transportation cost considering both the actual cost c_{ij} and the dual prices from the master problem. If the reduced cost is negative, it indicates that the new pattern can potentially reduce the total cost in the master problem and should therefore be added to the set of patterns (equations 18 to 20).

$$Minimize: \sum_{i\in I} \sum_{j\in J} (c_{ij} - \pi_j - \mu_i) \times x_{ij} \quad (18)$$

$$\sum_{j\in J} x_{ij} \leq s_i \quad \forall i \in I \quad (19)$$

$$x_{ij} \geq 0 \quad \forall i \in I, \quad \forall j \in J \quad (20)$$

If the reduced cost in the subproblem is non-negative, then no improving column exists, and the current solution of the master problem is optimal (equation 21).

$$\sum_{i\in I} \sum_{j\in J} (c_{ij} - \pi_j - \mu_i) \times x_{ij} \geq 0 \quad (21)$$

Table 4 presents the Python code that solves the transportation optimization problem using a column generation approach. This code illustrates how to leverage the PuLP library in Python to iteratively solve a linear optimization problem aimed at minimizing the total transportation cost from multiple origins to multiple destinations, subject to capacity and demand constraints. The code begins by importing necessary libraries including PuLP, NumPy, and time for runtime measurement. It defines the input data, specifying the transportation costs between origins and destinations, the capacity at each origin, and the demand at each destination. Two initial shipment patterns are created as starting points for the master problem: one where origin 'a' satisfies all demand, and another where origin 'b' does. The algorithm then

enters an iterative process. In each iteration, it first solves the master problem, which determines how many times each shipment pattern should be used to satisfy all demands without exceeding origin capacities, aiming to minimize total cost. After solving the master problem, the dual prices associated with demand and capacity constraints are extracted. Using these dual prices, the subproblem generates a new shipment pattern by solving a reduced cost minimization problem. If the subproblem finds a new pattern with negative reduced cost, this pattern is added to the master problem's set of columns for the next iteration. This process repeats until no new improving columns can be found, indicating that the optimal solution has been reached. Throughout the iterations, the code prints the current objective value, dual prices, and reduced cost of the new pattern. At the end, it outputs the final solution, showing which patterns are used and how many times, as well as the total minimized transportation cost. The code also reports the total runtime of the optimization process.

Table 4. Implementation of the Column Generation Algorithm for the Transportation Problem with Python.

```
# Python
# importing libraries
import pulp
import numpy as np
import time

# Input data
cost = {
    ('a', 1): 8, ('a', 2): 6, ('a', 3): 3,
    ('b', 1): 2, ('b', 2): 4, ('b', 3): 9,
}
capacity = {'a': 70, 'b': 40}
demand = {1: 40, 2: 35, 3: 25}
origins = ['a', 'b']
destinations = [1, 2, 3]

# Column generation algorithm
patterns = []        # List of current patterns (columns)
pattern_costs = []   # Cost of each pattern

# Initial patterns to start the master problem
initial_patterns = [
    # Pattern where 'a' satisfies all demand
    {('a', 1): 40, ('a', 2): 35, ('a', 3): 25, ('b', 1): 0, ('b', 2): 0, ('b', 3): 0},
    # Pattern where 'b' satisfies all demand
    {('a', 1): 0, ('a', 2): 0, ('a', 3): 0, ('b', 1): 40, ('b', 2): 35, ('b', 3): 25},
]

patterns.extend(initial_patterns)
pattern_costs = [
    sum(cost[(i,j)] * p[(i,j)] for i in origins for j in destinations)
    for p in patterns
]

# Start timer
start_time = time.time()

iteration = 0
while True:
    print(f"\nIteration {iteration}")
    iteration += 1

    # 🔷 Master Problem (MP)
    mp = pulp.LpProblem("MasterProblem", pulp.LpMinimize)
    lambdas = [pulp.LpVariable(f"lambda_{k}", lowBound=0, cat='Integer') for k in range(len(patterns))]

    # Objective function
    mp += pulp.lpSum([pattern_costs[k] * lambdas[k] for k in range(len(patterns))])

    # Demand constraints
    for j in destinations:
        mp += pulp.lpSum([patterns[k][(i,j)] * lambdas[k] for k in range(len(patterns)) for i in origins]) >= demand[j], f"demand_{j}"

    # Capacity constraints for each origin
    for i in origins:
        mp += pulp.lpSum([sum(patterns[k][(i,j)] for j in destinations) * lambdas[k] for k in range(len(patterns))]) <= capacity[i], f"capacity_{i}"

    mp.solve()
    print("MP Objective =", pulp.value(mp.objective))

    # Extract dual prices (shadow prices)
    dual_demand = {j: mp.constraints[f"demand_{j}"].pi for j in destinations}
    dual_supply = {i: mp.constraints[f"capacity_{i}"].pi for i in origins}
    print("Dual prices (demand):", dual_demand)
    print("Dual prices (supply):", dual_supply)

    # 🔶 Subproblem (SP) – generate new column with negative reduced cost
    sp = pulp.LpProblem("SubProblem", pulp.LpMinimize)
    x = pulp.LpVariable.dicts("x", [(i,j) for i in origins for j in destinations], lowBound=0, cat='Integer')

    # Reduced cost objective function
    sp += pulp.lpSum([
        (cost[(i,j)] - dual_demand[j] - dual_supply[i]) * x[(i,j)]
        for i in origins for j in destinations
    ])

    # Capacity constraints for subproblem
    for i in origins:
        sp += pulp.lpSum([x[(i,j)] for j in destinations]) <= capacity[i]

    sp.solve()
    reduced_cost = pulp.value(sp.objective)
```

```
    print("Reduced cost =", reduced_cost)

    # Stop if no negative reduced cost column is found
    if reduced_cost >= -1e-5:
        print("🛑 No negative reduced cost column — optimal solution found.")
        break

    # Add new column to the master problem
    new_pattern = {(i,j): int(x[(i,j)].varValue) for i in origins for j in destinations}
    new_cost = sum(cost[(i,j)] * new_pattern[(i,j)] for i in origins for j in destinations)

    patterns.append(new_pattern)
    pattern_costs.append(new_cost)
    print("✅ New column added with cost:", new_cost)

# Final result
print("\n======= FINAL SOLUTION =======")
for idx, var in enumerate(lambdas):
    if var.varValue > 0:
        print(f"Pattern {idx} used {var.varValue} times")
        print(patterns[idx])
print("Total cost =", pulp.value(mp.objective))

# Report runtime
end_time = time.time()
elapsed = end_time - start_time
print(f"\nTotal runtime: {elapsed:.4f} seconds")
```

```
Iteration 0
MP Objective = 541.0
Dual prices (demand): {1: 0.0, 2: 0.0, 3: 24.2}
Dual prices (supply): {'a': 0.0, 'b': -1.6}
Reduced cost = -2028.0
✅ New column added with cost: 570

Iteration 1
MP Objective = 541.0
Dual prices (demand): {1: 15.125, 2: 0.0, 3: 0.0}
Dual prices (supply): {'a': 0.0, 'b': -1.6}
Reduced cost = -959.75
✅ New column added with cost: 640

Iteration 2
MP Objective = 541.0
Dual prices (demand): {1: 0.0, 2: 17.285714, 3: 0.0}
Dual prices (supply): {'a': 0.0, 'b': -1.6}
Reduced cost = -1257.4285399999999
✅ New column added with cost: 580

Iteration 3
MP Objective = 540.999995665
Dual prices (demand): {1: 6.4, 2: 5.8545455, 3: 5.7636364}
Dual prices (supply): {'a': 0.0, 'b': -1.6}
Reduced cost = -305.45454800000005
✅ New column added with cost: 290

Iteration 4
MP Objective = 437.72727219999996
Dual prices (demand): {1: 5.8181818, 2: 5.2727273, 3: 0.81818182}
Dual prices (supply): {'a': 0.0, 'b': 0.0}
Reduced cost = -152.72727199999997
✅ New column added with cost: 80

Iteration 5
MP Objective = 403.62068915000003
Dual prices (demand): {1: 7.0344828, 2: 7.1034483, 3: 3.0}
Dual prices (supply): {'a': 0.0, 'b': -5.0344828}
Reduced cost = -77.24138100000002
✅ New column added with cost: 420

Iteration 6
MP Objective = 365.0000006
Dual prices (demand): {1: 4.0, 2: 6.0, 3: 3.0}
Dual prices (supply): {'a': 0.0, 'b': -2.0}
Reduced cost = 0.0
🛑 No negative reduced cost column — optimal solution found.

======= FINAL SOLUTION =======
Pattern 5 used 0.35714286 times
{('a', 1): 0, ('a', 2): 0, ('a', 3): 70, ('b', 1): 40, ('b', 2): 0, ('b', 3): 0}
Pattern 6 used 0.64285714 times
{('a', 1): 0, ('a', 2): 0, ('a', 3): 0, ('b', 1): 40, ('b', 2): 0, ('b', 3): 0}
Pattern 7 used 0.5 times
{('a', 1): 0, ('a', 2): 70, ('a', 3): 0, ('b', 1): 0, ('b', 2): 0, ('b', 3): 0}
Total cost = 365.0000006

Total runtime: 0.1248 seconds
```

The column generation algorithm converged to an optimal solution in 7 iterations with a total runtime of approximately 0.13 seconds. Throughout the iterations, new shipment patterns with negative reduced costs were added, progressively improving the master problem's objective value. The final solution achieved a minimum total transportation cost of 365 by using three key patterns with fractional values: Pattern 5 was used 0.3571 times, Pattern 6 was used 0.6429 times, and Pattern 7 was used 0.5 times. These patterns involve shipments primarily from origin 'b' to destination 1 and from origin 'a' to destinations 2 and 3, ensuring all demand and capacity constraints were satisfied optimally. Based on this mix, the total amount transported between origins and destinations is as follows:

- From origin 'a' to destination 2: 35 units, and to destination 3: 25 units
- From origin 'b' to destination 1: 40 units

Table 5 presents the Python code that uses PuLP's default solver to solve the problem, and no custom algorithm implementation is applied. The code starts by importing the necessary libraries, including PuLP, NumPy, and the time module. It then defines the

objective function coefficients (c), the constraint matrix (A), and the right-hand side of the constraints (b). Next, the code creates the optimization problem using PuLP's LpProblem class, sets the objective function, and adds constraints to the problem. The objective function is defined as the dot product of the coefficients (c) and the decision variables (x). The code records the start time before solving the problem and the end time after solving it to measure the runtime of the optimization. The runtime is the difference between these two times. Finally, the code prints the values of the decision variables (x) and the optimal value of the objective function. It also prints the optimization's runtime. Because of the limitations of the Pulp library and other libraries, the ability to report the number of iterations is not available.

Table 5. Python code to solve the transportation problem.

```
# Python
# importing libraries
!pip install pulp
import numpy as np
from pulp import LpProblem, LpVariable, LpMinimize
import time

# Create the objective function coefficients
c = np.array([8, 6, 3, 2, 4, 9, 0, 0, 0, 0, 0, M, M, M])

# Create the constraint matrix
A = np.array([[1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0],
              [1, 0, 0, 1, 0, 0, 0, 0, -1, 0, 0, 1, 0, 0],
              [0, 1, 0, 0, 1, 0, 0, 0, 0, -1, 0, 0, 1, 0],
              [0, 0, 1, 0, 0, 1, 0, 0, 0, 0, -1, 0, 0, 1]])

# Create the right-hand side of the constraints
b = np.array([70, 40, 40, 35, 25])

# Python
# Create the problem
problem = LpProblem("Column Generation", LpMinimize)

# Create the variables
x = [LpVariable(f"x{i+1}", lowBound=0) for i in range(14)]

# Set the objective function
problem += np.dot(c, x), "Objective Function"

# Add the constraints
for i in range(len(b)):
    problem += np.dot(A[i], x) == b[i]

# Measure the runtime
start_time = time.time()

# Solve the problem
problem.solve()

end_time = time.time()
runtime = end_time - start_time

# Print the results
for v in problem.variables():
    print(f"{v.name}: {v.value()}")

print(f"Optimal value: {problem.objective.value()}")
print(f"Runtime: {runtime:.6f} seconds")
```

```
Requirement already satisfied: pulp in
/usr/local/lib/python3.10/dist-packages (2.8.0)
x1: 0.0
x10: 0.0
x11: 0.0
x12: 0.0
x13: 0.0
x14: 0.0
x2: 35.0
x3: 25.0
x4: 40.0
x5: 0.0
x6: 0.0
x7: 10.0
x8: 0.0
x9: 0.0
Optimal value: 365.0
Runtime: 0.007255 seconds
```

According to the provided explanations and the Python code, the solution to the problem resulted in the following values: $x_{A2} = 35$, $x_{A3} = 25$, $x_{B1} = 40$, and $s_1 = 10$, with the remaining variables being equal to zero. In fact, it can be said that there are 10 units of excess capacity at point A. The Python code solved this problem in 0.007 seconds. Additionally, the objective function value was 365 units.

To verify the validity of the provided codes, Microsoft Excel Solver was used to solve the problem again, and the results can be seen in Table 6, which confirms the correctness of the Python code. Microsoft Excel Solver solved this problem in 0.016 seconds and in 6 iterations.

Table 6. Solving the transportation problem with Microsoft Excel Solver.

	Variables															
	x_{a1}	x_{a2}	x_{a3}	x_{b1}	x_{b2}	x_{b3}	s_1	s_2	e_1	e_2	e_3	a_1	a_2	a_3	Left-hand Side	Right-hand Side
Constraint 1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	70	70
Constraint 2	0	0	0	1	1	1	0	1	0	0	0	0	0	0	40	40
Constraint 3	1	0	0	1	0	0	0	0	-1		0	1	0	0	40	40
Constraint 4	0	1	0	0	1	0	0	0	0	-1	0	0	1	0	35	35
Constraint 5	0	0	1	0	0	1	0	0	0	0	-1	0	0	1	25	25
Variables	0	35	25	40	0	0	10	0	0	0	0	0	0	0	Z	
Objective Function	8	6	3	2	4	9	0	0	0	0	0	1000	1000	1000	365	

5.2. Cutting stock problem

Section 5.1 demonstrated that the column generation algorithm was able to successfully solve the transportation network problem, validating its effectiveness. This supplementary problem, introduced in Section 4.2, was used to further assess and illustrate the algorithm's performance. Considering equations 9 to 13 that described the cutting stock problem, and based on the explanations of the column generation method in Section 3, the master problem (the restricted master problem) is formed in equations 22 to 24, which is a basic solution for the original problem.

$$minimum: x_1 + x_2 + x_3 \quad (22)$$

$$S.t.$$

$$\begin{bmatrix}3\\0\\0\end{bmatrix}x_1 + \begin{bmatrix}0\\2\\0\end{bmatrix}x_2 + \begin{bmatrix}0\\0\\2\end{bmatrix}x_3 = \begin{bmatrix}80\\50\\100\end{bmatrix} \quad (23)$$

$$x_1, x_2, x_3 \geq 0 \quad (24)$$

Equations 25 to 29 form the subproblem, using the created master problem as the basis. In fact, with the help of the coefficient matrix of the basic variables in the objective function and the constraint matrix of the basic variables, this subproblem of maximization is formed, which aims to search for and find the variable that has the greatest effect on reducing the original objective function. Equation 28 reveals that a constraint (stock total length) exists in this subproblem, leading to a feasible region for our problem. This problem can easily be solved using the branch and bound algorithm, as detailed in the book "Operations Research: Applications and Algorithms" [14].

$$B_0 = \begin{bmatrix}300\\020\\002\end{bmatrix} \quad (25)$$

$$c_{BV} = [1\ 1\ 1] \quad (26)$$

$$maximum: c_{BV}B_0^{-1}\begin{bmatrix}a_4\\a_6\\a_7\end{bmatrix} - 1 = \frac{1}{3}a_4 + \frac{1}{2}a_6 + \frac{1}{2}a_7 - 1 \quad (27)$$

$$S.t.$$

$$4a_4 + 6a_6 + 7a_7 \leq 15 \quad (28)$$

$$a_4, a_6, a_7 \geq 0\ and\ integer \quad (29)$$

Solving this subproblem, the solutions a_4, a_6, and a_7 are 2, 0, and 1, respectively, which refers to the pattern x4. The objective function value is 7/6 -1 = 1/6, which is a positive value, and the variable, by being added, causes a decrease in the objective function of the master problem. After that, equations 30 to 32 are updated and displayed as the master problem. The subproblem is again formed based on the master problem (equations 33 to 35, with the variable x_4 entered instead of x1 in the subproblem (ratio test)). Using the branch and bound algorithm to solve this problem reveals that both variables x_5 and x_6, if entered into the problem, result in an objective function equal to 0, indicating they have no reducing effect on the master problem. In fact, the problem is solved in a single iteration. And the solution of the master problem (the restricted master problem) is x_1 = 0, x_2 = 25, x_3 = 30, x_4 = 40, and the objective function value is 95, which is also the solution to the original problem.

$$minimum: x_1 + x_2 + x_3 + x_4 \quad (30)$$

$$\begin{bmatrix}3\\0\\0\end{bmatrix} x_1 + \begin{bmatrix}0\\2\\0\end{bmatrix} x_2 + \begin{bmatrix}0\\0\\2\end{bmatrix} x_3 + \begin{bmatrix}2\\0\\1\end{bmatrix} x_4 = \begin{bmatrix}80\\50\\100\end{bmatrix} \quad (31)$$

$$x_1, x_2, x_3, x_4 \geq 0 \quad (32)$$

$$maximum: \frac{1}{4}a_4 + \frac{1}{2}a_6 + \frac{1}{2}a_7 - 1 \quad (33)$$

$$4a_4 + 6a_6 + 7a_7 \leq 15 \quad (34)$$

$$a_4, a_6, a_7 \geq 0 \; and \; integer \quad (35)$$

Table 7's Python code implements a column generation algorithm to solve a cutting stock problem. The code begins by importing the necessary libraries, including gurobipy for solving linear programming and integer programming problems, numpy for numerical operations, and math for mathematical functions. It then defines an instance class to represent the problem instance, which includes order lengths, order demands, and the roll length.

The generate_initial_patterns() function defines six initial cutting patterns and prints them, along with their values. These initial patterns serve as the starting point for the column generation algorithm. The define_master_problem() function defines the master problem as a linear programming formulation. The master problem entails determining the number of times to utilize each cutting pattern to fulfill orders, all while adhering to demand constraints. On the other hand, the define_subproblem() function defines the subproblem as a knapsack problem, with the goal of finding a new cutting pattern that can improve the master problem's solution.

The column_generation() function is the heart of the algorithm. It iteratively solves the master problem, obtains the dual variables, and solves the subproblem. If the subproblem's objective value is less than 1 (plus a small tolerance), the algorithm has found a new cutting pattern that can improve the solution, and it adds this pattern to the master problem. The function continues the process until it meets the termination condition, at which point it prints the final solution and the total number of rolls used. Finally, the code creates a new instance of the problem with the given order lengths and demands, summarizes the instance, and runs the column generation algorithm, storing the objective function values in the history variable.

Table 7. Python code to solve the cutting stock problem.

```
# Python
# importing libraries
!pip install gurobipy
from gurobipy import GRB
import gurobipy as gp
import numpy as np
import math

# Define a class to represent the instance of the problem
class Instance(object):

    def __init__(self, order_lengths, order_demands):
        # Initialize instance attributes
        self.n = len(order_lengths)  # Number of orders
        self.order_lens = order_lengths  # Length of each order
        self.demands = order_demands  # Demand for each order
        self.m = np.sum(self.demands)  # Total demand
        self.roll_len = 15  # Length of the roll

    def summarize(self):
        # Summarize the problem instance
        print("Problem instance with ", self.n, " orders and ", self.m,
"rolls")
        print("-"*47)
        print("\nOrders:\n")
        for i, order_len in enumerate(self.order_lens):
            print("\tOrder ", i, ": length= ", order_len, " demand=",
self.demands[i])
        print("\nRoll Length: ", self.roll_len)

# Function to generate initial cutting patterns
def generate_initial_patterns(ins_:Instance):
    patterns = []

    # Define initial cutting patterns
    pattern1 = [3, 0, 0]
    pattern2 = [0, 2, 0]
    pattern3 = [0, 0, 2]
    pattern4 = [2, 0, 1]
    pattern5 = [2, 1, 0]
    pattern6 = [0, 1, 1]

    # Add patterns to the list
    patterns.append(pattern1)
    patterns.append(pattern2)
    patterns.append(pattern3)
    patterns.append(pattern4)
    patterns.append(pattern5)
    patterns.append(pattern6)

    # Print the patterns and their values
    for i, pattern in enumerate(patterns):
        print(f"Pattern {i}: {pattern}")
        print(f"Value: {sum(pattern)}")
        print("--------------------")

    return patterns

# Function to define the master problem (linear programming
formulation)
def define_master_problem(ins_:Instance, patterns):
```

```
    n_pattern = len(patterns)
    pattern_range = range(n_pattern)
    order_range = range(ins_.n)
    patterns = np.array(patterns, dtype=int)
    master_problem = gp.Model("master problem")

    # Decision variables (lambda variables)
    lambda_ = master_problem.addVars(pattern_range,
                                     vtype=GRB.CONTINUOUS,
                                     obj=np.ones(n_pattern),
                                     name="lambda")

    # Direction of optimization (minimization)
    master_problem.modelSense = GRB.MINIMIZE

    # Demand satisfaction constraint
    for i in order_range:
        master_problem.addConstr(sum(patterns[p,i]*lambda_[p] for
p in pattern_range) == ins_.demands[i],
                                 "Demand[%d]" %i)

    # Solve the master problem
    return master_problem

# Function to define the subproblem (knapsack problem)
def define_subproblem(ins_:Instance, duals):
    order_range = range(ins_.n)
    subproblem = gp.Model("subproblem")

    # Decision variables (x variables)
    x = subproblem.addVars(order_range,
                           vtype=GRB.INTEGER,
                           obj=duals,
                           name="x")

    # Direction of optimization (maximization)
    subproblem.modelSense = GRB.MAXIMIZE

    # Length constraint (knapsack constraint)
    subproblem.addConstr(sum(ins_.order_lens[i] * x[i] for i in
order_range) <= ins_.roll_len)

    return subproblem

# Function to print the solution of the master problem
def print_solution(master, patterns):
    use = [math.ceil(i.x) for i in master.getVars()]
    for i, p in enumerate(patterns):
        if use[i]>0:
            print('Pattern ', i, ': how often we should cut: ', use[i])
            print('----------------------')
            for j,order in enumerate(p):
                if order >0:
                    print('order ', j, ' how much: ', order)
            print()

# Column generation algorithm
def column_generation(ins_:Instance):
    patterns = generate_initial_patterns(ins_)
    objVal_history = []

    while True:
        # Define and solve the master problem
        master_problem = define_master_problem(ins_, patterns)
        master_problem.optimize()
        objVal_history.append(master_problem.objVal)

        # Obtain dual variables from the master problem
        dual_variables = np.array([constraint.pi for constraint in
master_problem.getConstrs()])

        # Define and solve the subproblem
        subproblem = define_subproblem(ins_, dual_variables)
        subproblem.optimize()

        # Check termination condition
        if subproblem.objVal < 1 + 1e-2:
            break
        patterns.append([i.x for i in subproblem.getVars()])

    # Print the solution
    print_solution(master_problem, patterns)
    print('Total number of rolls used: ', int(np.array([math.ceil(i.x)
for i in master_problem.getVars()]).sum()))
    return objVal_history

# New instance details
order_lengths = [4, 6, 7]
order_demands = [80, 50, 100]

# Create an instance object
instance = Instance(order_lengths, order_demands)
instance.summarize()

# Run the column generation algorithm and store the objective
function values
history = column_generation(instance)
```

```
Orders:

        Order  0 : length=  4  demand= 80
        Order  1 : length=  6  demand= 50
        Order  2 : length=  7  demand= 100

Roll Length:  15
Pattern 0: [3, 0, 0]
Value: 3
----------
Pattern 1: [0, 2, 0]
Value: 2
----------
Pattern 2: [0, 0, 2]
Value: 2
----------
Pattern 3: [2, 0, 1]
Value: 3
----------
Pattern 4: [2, 1, 0]
Value: 3
----------
Pattern 5: [0, 1, 1]
Value: 2
```

```
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   7.4994000e+01   5.001500e+00   0.000000e+00     0s
       1   9.5000000e+01   0.000000e+00   0.000000e+00     0s

Optimal solution found (tolerance 1.00e-04)
Pattern  1 : how often we should cut:  25
----------------------
order  1  how much:  2

Pattern  2 : how often we should cut:  30
----------------------
order  2  how much:  2

Pattern  3 : how often we should cut:  40
----------------------
order  0  how much:  2
order  2  how much:  1

Total number of rolls used:  95
```

According to the provided explanations and the Python code, the solution to the cutting stock problem resulted in the following values: $x_2 = 25$, $x_3 = 30$, and $x_4 = 40$, with the remaining variables being equal to zero and the objective function value being 95 units. In one iteration and almost zero seconds, the Python code solved the problem.

To verify the validity of the provided code, Microsoft Excel Solver was used to solve the problem again, and the results can be seen in Table 8, which confirms the correctness of the Python code. The problem was solved in 0.016 seconds and 4 iterations using the Microsoft Excel Solver with the Simplex algorithm.

Table 8. Solving the cutting stock problem with Microsoft Excel Solver.

	Variables							
	x_1	x_2	x_3	x_4	x_5	x_6	Left-hand Side	Right-hand Side
Constraint 1	3	0	0	2	2	0	80	80
Constraint 2	0	2	0	0	1	1	50	50
Constraint 3	0	0	2	1	0	1	100	100
Variables	0	25	30	40	0	0	Z	
Objective Function	1	1	1	1	1	1	95	

6. Conclusion

The presented research delves into the application of the column generation algorithm to address two fundamental optimization problems: transportation network optimization and the cutting stock problem.

Although the transportation network optimization problem involved inherent complexities, the column generation algorithm proved to be effective in solving it. The algorithm successfully reached the optimal solution in 7 iterations. In comparison, Microsoft Excel's built-in solver found the solution in 6 iterations, and both Excel and PuLP's default solver performed slightly faster due to the small size and simplicity of the network. However, despite this, the column generation method remains a promising approach—especially for large-scale problems—where its iterative structure and ability to handle vast numbers of variables can offer significant advantages over traditional methods such as the simplex algorithm.

The study showcased the efficacy of the column generation algorithm in addressing the cutting stock problem. By iteratively generating and adding cutting patterns to the master problem, the algorithm efficiently minimized waste and optimized stock cutting, resulting in a solution with minimal computational resources and runtime. In the cutting stock problem, it was observed that the column generation algorithm only found the optimal solution in a single iteration, while the simplex method reached the solution in 4 iterations. Given the large real-world problems, the column generation algorithm is very intelligent and cost-effective, significantly reducing the computational cost. The results were validated through comparison with solutions obtained using

Microsoft Excel Solver, affirming the correctness and efficiency of the Python implementation.

In summary, while the column generation algorithm demonstrated both versatility and effectiveness in solving optimization problems, its applicability can vary depending on the structure and complexity of the specific problem. Further research and experimentation may be needed to explore alternative methods and enhance the algorithm's performance, particularly when addressing a wider range of optimization challenges in transportation and other complex domains.

References

[1] Chopra, S., & Meindl, P. (2020). Supply Chain Management: Strategy, Planning, and Operation. Pearson Education Limited.

[2] Banister, D. (2008). The sustainable mobility paradigm. Transport Policy, 15(2), 73–80.

[3] Litman, T. (2019). Evaluating public transit benefits and costs. Victoria Transport Policy Institute.

[4] Hultkrantz, L., & Lundberg, M. (2006). Cost-benefit analysis of transportation investments: recent developments in Sweden. Transport Reviews, 26(5), 593-611.

[5] Dantzig, G. B., & Wolfe, P. (1960). Decomposition principle for linear programs. Operations Research, 8(1), 101–111.

[6] Desaulniers, G., Desrosiers, J., & Solomon, M. M. (2005). Column generation. Springer Science & Business Media.

[7] Gilmore, P. C., & Gomory, R. E. (1961). A Linear Programming Approach to the Cutting Stock Problem. Operations Research, 9(6), 849–859.

[8] Desaulniers, G., Desrosiers, J., Ioachim, I., Solomon, M. M., Soumis, F., & Villeneuve, D. (1998). A unified framework for deterministic time constrained vehicle routing and crew scheduling problems. Transportation Science, 8-35.

[9] Barnhart, C., Cohn, A. M., Johnson, E. L., Klabjan, D., Nemhauser, G. L., & Vance, P. H. (2002). Airline crew scheduling. Handbook of transportation science, 517-560.

[10] Gendron, B., Crainic, T. G., & Frangioni, A. (1999). Multicommodity capacitated network design. Telecommunications network planning, 1-19.

[11] Lai, M. F., & Lo, H. K. (2004). Ferry service network design: optimal fleet size, routing, and scheduling. Transportation Research Part A: Policy and Practice, 38(4), 305-328.

[12] Melo, M. T., Nickel, S., & Saldanha-da-Gama, F. (2014). A Column Generation Approach for the Integrated Planning of Urban Delivery Services. European Journal of Operational Research.

[13] Desaulniers, G., et al. (2008). A Column Generation Approach to Line Planning in Public Transit. Transportation Science.

[14] Winston, W. L. (2004). *Operations Research: Applications and Algorithms* (4th ed.). Indiana University.

[15] Sallan, J. M., Lordan, O., & Fernandez, V. (2015). Modeling and solving linear programming with R.OmniaScience.