demo1

December 9, 2024

1 Use Gurobi Optimizer

[1]:
```
import gurobipy as gp

try:
    m = gp.Model("test")
    print("Gurobi license is working!")
except gp.GurobiError as e:
    print(f"Error: {e}")
```

```
Gurobi license is working!
```

[8]:
```
# Video Quality Optimization using Gurobi
import gurobipy as gp
from gurobipy import GRB
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

[15]:
```
def define_problem_data():
    # Resolution bandwidth requirements (Mbps)
    B = {
        '8K': 200,
        '4K': 45,
        '2K': 16,
        '1K': 8
    }

    # Quality scores for each resolution
    q = {
        '8K': 4,
        '4K': 3,
        '2K': 2,
        '1K': 1
    }

    # Client maximum bitrates
```

```python
    max_bitrate = {
        'A': 10,
        'B': 50,
        'C': 200,
        'D': 200,
        'E': 20,
        'F': 50,
        'G': 200,
        'H': 200
    }

    # Link capacities
    links = {
        ('Server', 'Router_C'): 300,
        ('Router_C', 'Router_A'): 250,
        ('Router_C', 'Router_B'): 250,
        ('Router_A', 'Client_A'): 200,
        ('Router_A', 'Client_B'): 200,
        ('Router_A', 'Client_C'): 200,
        ('Router_A', 'Client_D'): 200,
        ('Router_B', 'Client_E'): 200,
        ('Router_B', 'Client_F'): 200,
        ('Router_B', 'Client_G'): 200,
        ('Router_B', 'Client_H'): 200
    }

    # Define paths for each client
    paths = {
        'A': [('Server', 'Router_C'), ('Router_C', 'Router_A'), ('Router_A', 'Client_A')],
        'B': [('Server', 'Router_C'), ('Router_C', 'Router_A'), ('Router_A', 'Client_B')],
        'C': [('Server', 'Router_C'), ('Router_C', 'Router_A'), ('Router_A', 'Client_C')],
        'D': [('Server', 'Router_C'), ('Router_C', 'Router_A'), ('Router_A', 'Client_D')],
        'E': [('Server', 'Router_C'), ('Router_C', 'Router_B'), ('Router_B', 'Client_E')],
        'F': [('Server', 'Router_C'), ('Router_C', 'Router_B'), ('Router_B', 'Client_F')],
        'G': [('Server', 'Router_C'), ('Router_C', 'Router_B'), ('Router_B', 'Client_G')],
        'H': [('Server', 'Router_C'), ('Router_C', 'Router_B'), ('Router_B', 'Client_H')]
    }
```

```
    return B, q, max_bitrate, links, paths

class VideoOptimizationModel:
    def __init__(self):
        self.B, self.q, self.max_bitrate, self.links, self.paths = define_problem_data()
        self.clients = list(self.max_bitrate.keys())
        self.resolutions = list(self.B.keys())
        self.model = None
        self.x = None
        self.m = None
        self.b = None
        self.x_slack = None
        # Create the model right after initialization
        self.create_model()

    def create_model(self):
        """Create the optimization model and variables"""
        self.model = gp.Model("video_quality_optimization")

        # Create variables
        self.x = self.model.addVars(self.clients, self.resolutions, vtype=GRB.BINARY, name="x")
        self.m = self.model.addVars(self.links.keys(), self.resolutions, vtype=GRB.BINARY, name="m")
        self.b = self.model.addVars(self.links.keys(), vtype=GRB.CONTINUOUS, name="b")
        self.x_slack = self.model.addVars(self.clients, vtype=GRB.CONTINUOUS, name="x_slack")

    def set_objective(self, alpha=1.0, gamma=0.5, lambda_val=1000):
        """Set the optimization objective"""
        obj = -alpha * gp.quicksum(self.q[j] * self.x[i,j]
                                 for i in self.clients for j in self.resolutions)
        obj += -gamma * gp.quicksum(self.x[i,j]
                                  for i in self.clients for j in self.resolutions)
        obj += lambda_val * gp.quicksum(self.x_slack[i] for i in self.clients)

        self.model.setObjective(obj, GRB.MINIMIZE)

    def add_constraints(self):
        """Add all model constraints"""
        # User assignment constraint
        for i in self.clients:
```

```
            self.model.addConstr(gp.quicksum(self.x[i,j] for j in self.resolutions) +
                               self.x_slack[i] == 1)

        # User capability constraint
        for i in self.clients:
            for j in self.resolutions:
                if self.B[j] > self.max_bitrate[i]:
                    self.model.addConstr(self.x[i,j] == 0)

        # Link bandwidth usage
        for l, k in self.links:
            self.model.addConstr(self.b[l,k] ==
                               gp.quicksum(self.m[l,k,j] * self.B[j] for j in self.resolutions))

        # Link capacity constraint
        for (l,k) in self.links:
            self.model.addConstr(self.b[l,k] <= self.links[l,k])

        # Multicast logic constraint
        for i in self.clients:
            for j in self.resolutions:
                for l,k in self.paths[i]:
                    self.model.addConstr(self.m[l,k,j] >= self.x[i,j])

    def optimize(self):
        """Run the optimization"""
        self.model.optimize()

    def print_model_complexity(self):
        """Print model complexity information"""
        if self.model:
            print("\nModel Complexity Statistics:")
            print(f"Number of Variables: {self.model.NumVars}")
            print(f"- Binary Variables: {sum(1 for v in self.model.getVars() if v.vtype == GRB.BINARY)}")
            print(f"- Continuous Variables: {sum(1 for v in self.model.getVars() if v.vtype == GRB.CONTINUOUS)}")
            print(f"Number of Constraints: {self.model.NumConstrs}")
            print(f"Number of Nonzeros: {self.model.NumNZs}")
            print(f"Objective Sense: {'Minimization' if self.model.ModelSense == 1 else 'Maximization'}")

            # Print variable types statistics
            var_types = {}
            for v in self.model.getVars():
```

```python
                var_type = v.VarName.split('[')[0]
                var_types[var_type] = var_types.get(var_type, 0) + 1
            print("\nVariable Counts by Type:")
            for var_type, count in var_types.items():
                print(f"- {var_type}: {count}")

    def plot_network_with_utilization(self, df_links):
        """Plot network topology with link utilization and client resolutions"""
        # Create figure and axis objects with a single subplot
        fig, ax = plt.subplots(figsize=(15, 10))

        # Define positions for nodes
        pos = {
            'Server': (0.2, 0.8),
            'Router_C': (0.4, 0.6),
            'Router_A': (0.2, 0.4),
            'Router_B': (0.6, 0.4),
            'Client_A': (0, 0.2),
            'Client_B': (0.2, 0.2),
            'Client_C': (0.4, 0.2),
            'Client_D': (0.6, 0.2),
            'Client_E': (0.8, 0.2),
            'Client_F': (1.0, 0.2),
            'Client_G': (1.2, 0.2),
            'Client_H': (1.4, 0.2)
        }

        # Get client resolutions
        client_resolutions = {}
        for i in self.clients:
            for j in self.resolutions:
                if self.x[i,j].x > 0.5:
                    client_resolutions[i] = j

        # Create a color map for link utilization
        utilization_dict = {(row['From'], row['To']): row['Utilization (%)']
                            for _, row in df_links.iterrows()}

        # Define a color mapping function
        def get_color(util_pct):
            # Returns colors from green (0%) to red (100%)
            return plt.cm.RdYlGn_r(util_pct / 100)

        # Plot nodes
        for node, (x, y) in pos.items():
            if 'Client' in node:
```

```
                ax.plot(x, y, 'gs', markersize=20, label='Client' if node ==␣
↪'Client_A' else "")
                ax.text(x, y-0.02, node, ha='center')
                # Add both maximum bitrate and assigned resolution
                client_id = node.split('_')[1]
                resolution = client_resolutions.get(client_id, 'N/A')
                ax.text(x, y-0.05, f'Max: {self.max_bitrate[client_id]}␣
↪Mbps\nAssigned: {resolution}',
                        ha='center', fontsize=8)
            elif 'Router' in node:
                ax.plot(x, y, 'bo', markersize=20, label='Router' if node ==␣
↪'Router_A' else "")
                ax.text(x, y-0.02, node, ha='center')
            else:
                ax.plot(x, y, 'rd', markersize=20, label='Server')
                ax.text(x, y-0.02, node, ha='center')

        # Plot edges with utilization colors and labels
        for (l, k), util_pct in utilization_dict.items():
            x1, y1 = pos[l]
            x2, y2 = pos[k]

            # Draw the edge with color based on utilization
            ax.plot([x1, x2], [y1, y2], '-', color=get_color(util_pct),␣
↪linewidth=2)

            # Add capacity and utilization labels
            label = f'{round(util_pct, 1)}%\n({self.links[l,k]} Mbps)'
            if 'Router' in l and 'Client' in k:
                ax.text((x1 + x2)/2 + 0.02, (y1 + y2)/2, label, ha='left')
            else:
                ax.text((x1 + x2)/2, (y1 + y2)/2 + 0.02, label)

        # Add colorbar
        norm = plt.Normalize(0, 100)
        sm = plt.cm.ScalarMappable(cmap=plt.cm.RdYlGn_r, norm=norm)
        sm.set_array([])
        plt.colorbar(sm, ax=ax, label='Link Utilization (%)')

        # Add server capabilities
        ax.text(0.2, 0.85, 'Resolutions:', ha='left')
        ax.text(0.2, 0.83, 'r4: 4320p(8K) -- 200Mbps', ha='left')
        ax.text(0.2, 0.81, 'r3: 2160p(4K) -- 45Mbps', ha='left')
        ax.text(0.2, 0.79, 'r2: 1440p(2K) -- 16Mbps', ha='left')
        ax.text(0.2, 0.77, 'r1: 1080p(1K) -- 8Mbps', ha='left')
```

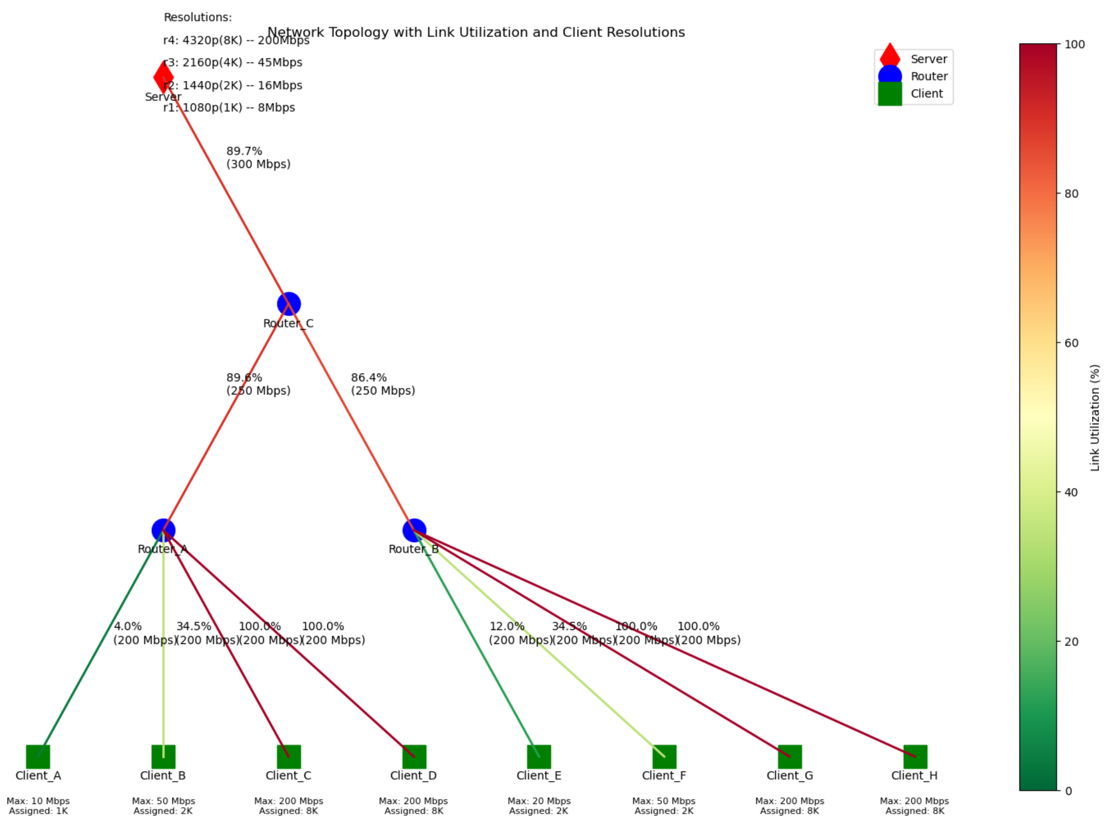
```
        ax.set_title('Network Topology with Link Utilization and Client␣
↪Resolutions')
        ax.legend()
        ax.axis('off')
        plt.tight_layout()
        plt.show()

    def print_results(self):
        """Print and visualize optimization results"""
        if self.model.status == GRB.OPTIMAL:
            # Print model complexity
            self.print_model_complexity()

            print("\nOptimal solution found!")

            # Create results DataFrame
            results = []
            for i in self.clients:
                for j in self.resolutions:
                    if self.x[i,j].x > 0.5:
                        results.append({
                            'Client': i,
                            'Resolution': j,
                            'Bandwidth': self.B[j]
                        })

            df_results = pd.DataFrame(results)
            print("\nClient assignments:")
            display(df_results)

            # Create link usage DataFrame
            link_usage = []
            for (l,k) in self.links:
                link_usage.append({
                    'From': l,
                    'To': k,
                    'Usage (Mbps)': round(self.b[l,k].x, 2),
                    'Capacity (Mbps)': self.links[l,k],
                    'Utilization (%)': round(self.b[l,k].x / self.links[l,k] *␣
↪100, 2)
                })

            df_links = pd.DataFrame(link_usage)
            print("\nLink bandwidth usage:")
            display(df_links)

            # Plot network topology with utilization
```

```python
            self.plot_network_with_utilization(df_links)
        else:
            print("No optimal solution found")

def run_optimization():
    # Create model instance
    model = VideoOptimizationModel()

    # Set objective with weights
    model.set_objective(alpha=1.0, gamma=0.5, lambda_val=1000)

    # Add constraints
    model.add_constraints()

    # Run optimization
    model.optimize()

    # Print results
    model.print_results()

if __name__ == "__main__":
    run_optimization()
```

```
Gurobi Optimizer version 10.0.1 build v10.0.1rc0 (win64)

CPU model: 13th Gen Intel(R) Core(TM) i7-1360P, instruction set [SSE2|AVX|AVX2]
Thread count: 12 physical cores, 16 logical processors, using up to 16 threads

Optimize a model with 133 rows, 95 columns and 305 nonzeros
Model fingerprint: 0x879b14b3
Variable types: 19 continuous, 76 integer (76 binary)
Coefficient statistics:
  Matrix range     [1e+00, 2e+02]
  Objective range  [2e+00, 1e+03]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 3e+02]
Found heuristic solution: objective 8000.0000000
Presolve removed 121 rows and 83 columns
Presolve time: 0.00s
Presolved: 12 rows, 12 columns, 29 nonzeros
Found heuristic solution: objective -27.0000000
Variable types: 0 continuous, 12 integer (12 binary)

Root relaxation: cutoff, 3 iterations, 0.00 seconds (0.00 work units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time
```

```
    0     0     cutoff    0       -27.00000  -27.00000  0.00%     -    0s

Explored 1 nodes (3 simplex iterations) in 0.02 seconds (0.00 work units)
Thread count was 16 (of 16 available processors)

Solution count 2: -27 8000
No other solutions better than -27

Optimal solution found (tolerance 1.00e-04)
Best objective -2.700000000000e+01, best bound -2.700000000000e+01, gap 0.0000%

Model Complexity Statistics:
Number of Variables: 95
- Binary Variables: 76
- Continuous Variables: 19
Number of Constraints: 133
Number of Nonzeros: 305
Objective Sense: Minimization

Variable Counts by Type:
- x: 32
- m: 44
- b: 11
- x_slack: 8

Optimal solution found!

Client assignments:

  Client Resolution  Bandwidth
0      A         1K          8
1      B         2K         16
2      C         8K        200
3      D         8K        200
4      E         2K         16
5      F         2K         16
6      G         8K        200
7      H         8K        200


Link bandwidth usage:

        From        To  Usage (Mbps)  Capacity (Mbps)  Utilization (%)
0     Server  Router_C         269.0              300            89.67
1   Router_C  Router_A         224.0              250            89.60
2   Router_C  Router_B         216.0              250            86.40
3   Router_A  Client_A           8.0              200             4.00
4   Router_A  Client_B          69.0              200            34.50
5   Router_A  Client_C         200.0              200           100.00
```

6	Router_A	Client_D	200.0	200	100.00
7	Router_B	Client_E	24.0	200	12.00
8	Router_B	Client_F	69.0	200	34.50
9	Router_B	Client_G	200.0	200	100.00
10	Router_B	Client_H	200.0	200	100.00



2 Complexity Analysis in Optimization Problem

In academic literature, the complexity of such optimization problems is typically characterized along the following dimensions:

2.1 Variable Complexity

The scale of decision variables is $O(|C| \times |R| + |E| \times |R| + |E|)$, where: - C represents the set of clients - R denotes the set of resolutions - E signifies the set of edges

2.2 Constraint Complexity

The magnitude of constraints is $O(|C| + |C| \times |R| + |E| + |C| \times |R| \times |P|)$, where P represents the path length.

2.3 Problem Parameters

In our specific instance: - $|C| = 8$ (clients: A through H) - $|R| = 4$ (resolutions: 8K, 4K, 2K, 1K) - $|E| = 11$ (edges: Server→Router_C, Router_C→Router_A/B, Router_A/B→Clients) - $|P| = 3$ (maximum path length: three edges from server to each client)

2.4 Detailed Computation

2.4.1 Variable Count

$O(|C|\times|R|+|E|\times|R|+|E|)$ comprises: - $|C|\times|R| = 8\times4 = 32$ (variables x_{ij}) - $|E|\times|R| = 11\times4 = 44$ (variables $m_{lk,j}$) - $|E| = 11$ (variables b_{lk}) - $|C| = 8$ (slack variables x'_i)

Total variables: $32 + 44 + 11 + 8 = 95$

2.4.2 Constraint Count

$O(|C| + |C| \times |R| + |E| + |C| \times |R| \times |P|)$ comprises: - $|C| = 8$ (client assignment constraints) - $|C|\times|R| = 32$ (client capability constraints) - $|E| = 11$ (link capacity constraints) - $|C|\times|R|\times|P| = 8 \times 4 \times 3 = 96$ (multicast logic constraints)

Total constraints: $8 + 32 + 11 + 96 = 147$

3 Scalability Analysis (using gurobipy)

[22]:
```
import gurobipy as gp
from gurobipy import GRB
import pandas as pd
import matplotlib.pyplot as plt
import time
import numpy as np
from typing import Dict, List, Tuple

class ScalabilityAnalysis:
    def __init__(self):
        # Base video resolution and quality parameters
        self.B = {
            '8K': 200,
            '4K': 45,
            '2K': 16,
            '1K': 8
        }
        self.q = {
            '8K': 4,
            '4K': 3,
            '2K': 2,
            '1K': 1
        }
```

```python
    def generate_network(self, num_users: int) -> Tuple[Dict, Dict, Dict]:
        """
        Generate network topology based on number of users
        Returns: max_bitrate, links, paths
        """
        # Calculate required number of routers (each router supports 8 users)
        num_routers = max(1, (num_users + 7) // 8)

        # Generate user maximum bitrates (randomly assign different user␣
↪capabilities)
        max_bitrate = {}
        bitrate_options = [10, 50, 200]
        for i in range(num_users):
            client_id = chr(65 + i) if i < 26 else f'User_{i}'
            max_bitrate[client_id] = np.random.choice(bitrate_options)

        # Generate link capacities
        links = {}
        # Server to core router link
        core_capacity = num_users * 50  # Estimate required capacity
        links[('Server', 'Router_Core')] = core_capacity

        # Core to access router links
        for i in range(num_routers):
            router_id = f'Router_{i}'
            links[('Router_Core', router_id)] = core_capacity // num_routers

            # Access router to client links
            start_user = i * 8
            end_user = min((i + 1) * 8, num_users)
            for j in range(start_user, end_user):
                client_id = chr(65 + j) if j < 26 else f'User_{j}'
                links[(router_id, f'Client_{client_id}')] = 200

        # Generate paths
        paths = {}
        for i in range(num_users):
            client_id = chr(65 + i) if i < 26 else f'User_{i}'
            router_id = f'Router_{i // 8}'
            paths[client_id] = [
                ('Server', 'Router_Core'),
                ('Router_Core', router_id),
                (router_id, f'Client_{client_id}')
            ]

        return max_bitrate, links, paths
```

```python
    def plot_network_topology(self, links: Dict, client_resolutions: Dict =␣
↪None):
        """Plot network topology with optional client resolution information"""
        fig, ax = plt.subplots(figsize=(15, 10))

        # Create directed graph for visualization
        unique_nodes = set()
        for (src, dst) in links.keys():
            unique_nodes.add(src)
            unique_nodes.add(dst)

        # Calculate layout
        pos = {}
        servers = [n for n in unique_nodes if 'Server' in n]
        core_routers = [n for n in unique_nodes if 'Router_Core' in n]
        access_routers = [n for n in unique_nodes if 'Router_' in n and 'Core'␣
↪not in n]
        clients = [n for n in unique_nodes if 'Client_' in n]

        # Position calculations
        y_levels = {'Server': 0.9, 'Core': 0.7, 'Access': 0.5, 'Client': 0.2}

        # Position servers
        for i, node in enumerate(servers):
            pos[node] = (0.5, y_levels['Server'])

        # Position core routers
        for i, node in enumerate(core_routers):
            pos[node] = (0.5, y_levels['Core'])

        # Position access routers
        router_spacing = 1.0 / (len(access_routers) + 1)
        for i, node in enumerate(access_routers):
            pos[node] = ((i + 1) * router_spacing, y_levels['Access'])

        # Position clients
        clients_per_router = len(clients) / len(access_routers)
        current_client = 0
        for i, router in enumerate(access_routers):
            router_clients = clients[int(i * clients_per_router):int((i + 1) *␣
↪clients_per_router)]
            client_spacing = router_spacing / (len(router_clients) + 1)
            router_x = pos[router][0]
            start_x = router_x - router_spacing/2
            for j, client in enumerate(router_clients):
                pos[client] = (start_x + (j + 1) * client_spacing,␣
↪y_levels['Client'])
```

```python
        # Draw nodes
        node_colors = {'Server': 'red', 'Router': 'blue', 'Client': 'green'}
        for node in unique_nodes:
            x, y = pos[node]
            if 'Server' in node:
                color = node_colors['Server']
                marker = 's'
            elif 'Router' in node:
                color = node_colors['Router']
                marker = 'o'
            else:
                color = node_colors['Client']
                marker = '^'
            ax.scatter(x, y, c=color, marker=marker, s=200)

            # Add node labels
            if client_resolutions and 'Client_' in node:
                client_id = node.split('_')[1]
                resolution = client_resolutions.get(client_id, 'N/A')
                ax.text(x, y-0.05, f'{node}\n{resolution}', ha='center', va='top')
            else:
                ax.text(x, y-0.05, node, ha='center', va='top')

        # Draw edges
        for (src, dst), capacity in links.items():
            x1, y1 = pos[src]
            x2, y2 = pos[dst]
            ax.plot([x1, x2], [y1, y2], 'k-', alpha=0.5)
            # Add capacity labels
            mid_x = (x1 + x2) / 2
            mid_y = (y1 + y2) / 2
            ax.text(mid_x, mid_y, f'{capacity}Mbps', ha='center', va='center', bbox=dict(facecolor='white', alpha=0.7))

        # Add legend
        legend_elements = [
            plt.Line2D([0], [0], marker='s', color='w', label='Server', markerfacecolor='red', markersize=10),
            plt.Line2D([0], [0], marker='o', color='w', label='Router', markerfacecolor='blue', markersize=10),
            plt.Line2D([0], [0], marker='^', color='w', label='Client', markerfacecolor='green', markersize=10)
        ]
        ax.legend(handles=legend_elements, loc='upper right')
```

```
        plt.title('Network Topology')
        plt.grid(True, linestyle='--', alpha=0.3)
        plt.tight_layout()
        plt.show()

    def create_and_solve_model(self, num_users: int) -> Dict:
        """
        Create and solve optimization model for specific scale
        Returns solution statistics
        """
        try:
            # Generate network topology
            max_bitrate, links, paths = self.generate_network(num_users)
            clients = list(max_bitrate.keys())
            resolutions = list(self.B.keys())

            # Create model
            start_time = time.time()
            model = gp.Model("video_quality_optimization")
            model.setParam('OutputFlag', 0)

            # Create variables
            x = model.addVars(clients, resolutions, vtype=GRB.BINARY, name="x")
            m = model.addVars(links.keys(), resolutions, vtype=GRB.BINARY, name="m")
            b = model.addVars(links.keys(), vtype=GRB.CONTINUOUS, name="b")
            x_slack = model.addVars(clients, vtype=GRB.CONTINUOUS, name="x_slack")

            # Set objective function
            alpha, gamma, lambda_val = 1.0, 0.5, 1000
            obj = -alpha * gp.quicksum(self.q[j] * x[i,j] for i in clients for j in resolutions)
            obj += -gamma * gp.quicksum(x[i,j] for i in clients for j in resolutions)
            obj += lambda_val * gp.quicksum(x_slack[i] for i in clients)
            model.setObjective(obj, GRB.MINIMIZE)

            # Add constraints
            self._add_constraints(model, x, m, b, x_slack, clients, resolutions, links, paths, max_bitrate)

            # Solve and collect statistics
            setup_time = time.time() - start_time
            model.optimize()
            solve_time = model.Runtime
```

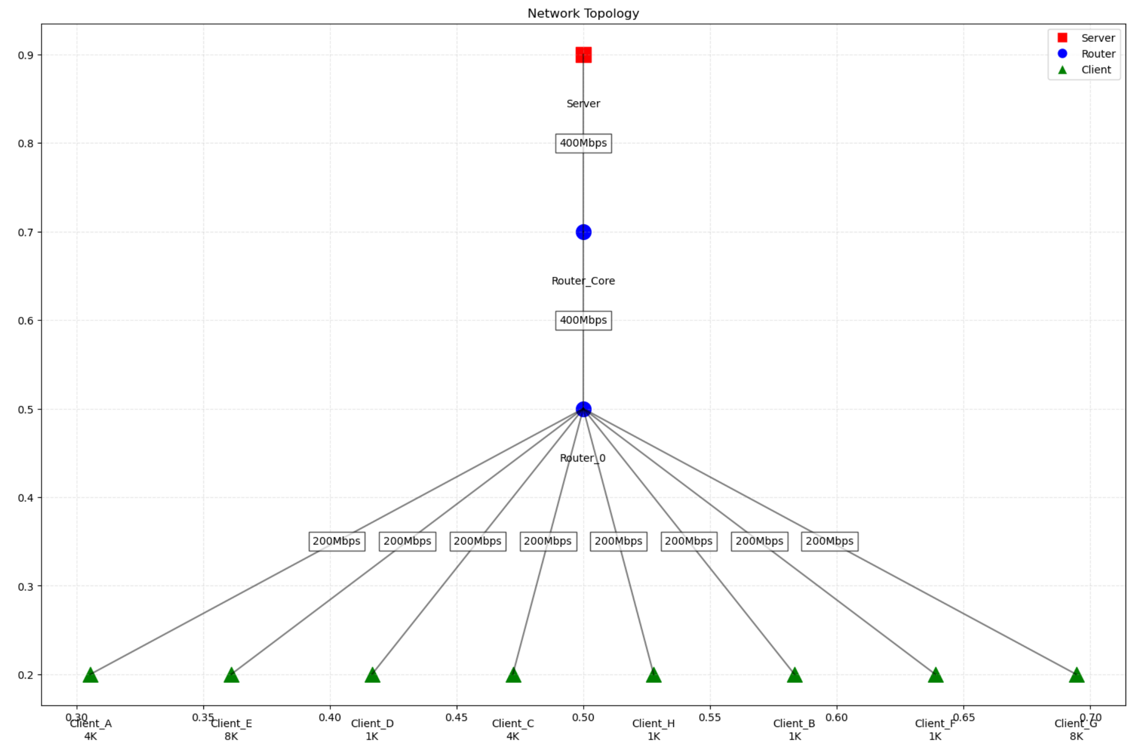
```
            # Plot network topology
            if model.status == GRB.OPTIMAL:
                client_resolutions = {}
                for i in clients:
                    for j in resolutions:
                        if x[i,j].x > 0.5:
                            client_resolutions[i] = j
                self.plot_network_topology(links, client_resolutions)

            return {
                'num_users': num_users,
                'num_variables': model.NumVars,
                'num_constraints': model.NumConstrs,
                'setup_time': setup_time,
                'solve_time': solve_time,
                'total_time': setup_time + solve_time,
                'objective_value': model.ObjVal if model.status == GRB.OPTIMAL else None,
                'model_status': model.status,
                'status': 'Success'
            }

        except Exception as e:
            return {
                'num_users': num_users,
                'num_variables': 0,
                'num_constraints': 0,
                'setup_time': 0,
                'solve_time': 0,
                'total_time': 0,
                'objective_value': None,
                'model_status': None,
                'status': 'Failed',
                'error': str(e)
            }

    def _add_constraints(self, model, x, m, b, x_slack, clients, resolutions, links, paths, max_bitrate):
        """Add all model constraints"""
        # User assignment constraint
        for i in clients:
            model.addConstr(gp.quicksum(x[i,j] for j in resolutions) + x_slack[i] == 1)

        # User capability constraint
        for i in clients:
            for j in resolutions:
```

```
                if self.B[j] > max_bitrate[i]:
                    model.addConstr(x[i,j] == 0)

        # Link bandwidth constraints
        for l, k in links:
            model.addConstr(b[l,k] == gp.quicksum(m[l,k,j] * self.B[j] for j in␣
↪resolutions))
            model.addConstr(b[l,k] <= links[l,k])

        # Multicast logic constraint
        for i in clients:
            for j in resolutions:
                for l,k in paths[i]:
                    model.addConstr(m[l,k,j] >= x[i,j])

    def run_scalability_analysis(self, user_scales: List[int]) -> pd.DataFrame:
        """
        Run performance analysis for different scales
        """
        results = []
        for num_users in user_scales:
            print(f"\nAnalyzing user count: {num_users}")
            stats = self.create_and_solve_model(num_users)
            results.append(stats)
            print(f"Solution status: {stats['status']}")
            if stats['status'] == 'Success':
                print(f"Completed - Total time: {stats['total_time']:.2f}s")
            else:
                print(f"Failed - Error: {stats.get('error', 'Unknown error')}")

        return pd.DataFrame(results)

    def plot_scalability_results(self, results: pd.DataFrame):
        """
        Plot performance analysis results
        """
        success_results = results[results['status'] == 'Success'].copy()

        if len(success_results) == 0:
            print("No successful solutions to plot")
            return

        fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(15, 12))

        # Variables growth
        ax1.plot(success_results['num_users'],␣
↪success_results['num_variables'], 'bo-')
```

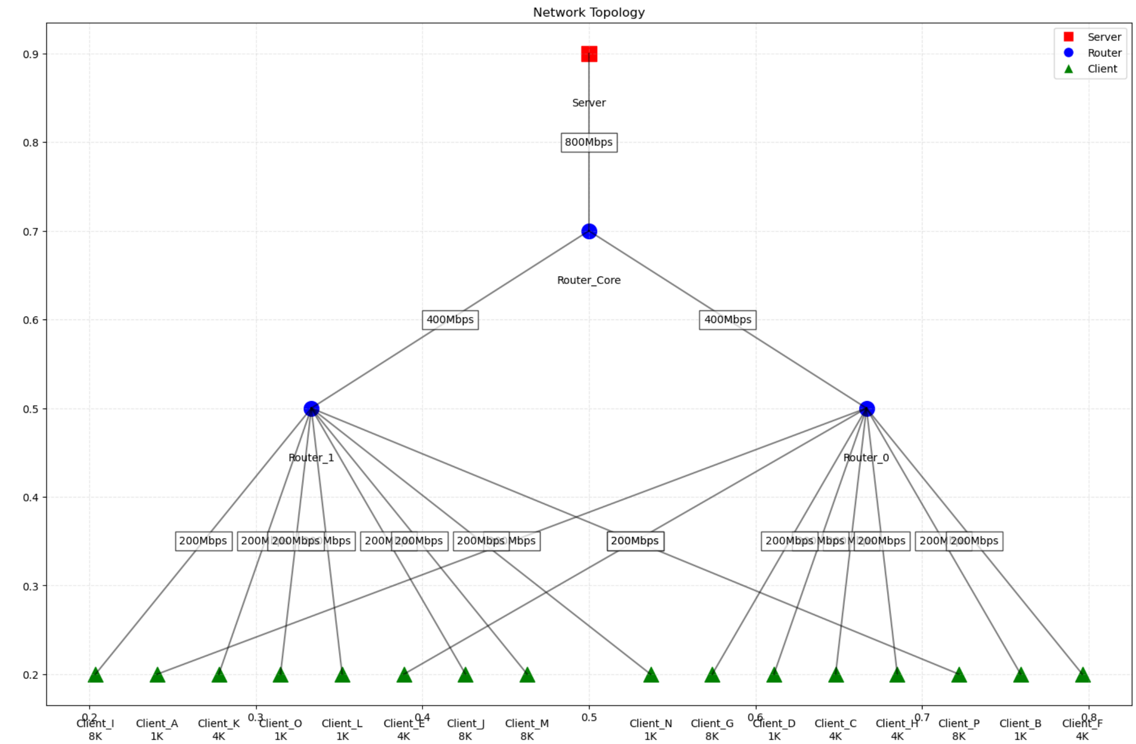
```
        ax1.set_title('Variables Growth')
        ax1.set_xlabel('Number of Users')
        ax1.set_ylabel('Number of Variables')
        ax1.grid(True)

        # Constraints growth
        ax2.plot(success_results['num_users'],␣
 ↪success_results['num_constraints'], 'ro-')
        ax2.set_title('Constraints Growth')
        ax2.set_xlabel('Number of Users')
        ax2.set_ylabel('Number of Constraints')
        ax2.grid(True)

        # Solution time growth
        ax3.plot(success_results['num_users'], success_results['solve_time'],␣
 ↪'go-')
        ax3.set_title('Solution Time Growth')
        ax3.set_xlabel('Number of Users')
        ax3.set_ylabel('Solution Time (seconds)')
        ax3.grid(True)

        # Total time growth
        ax4.plot(success_results['num_users'], success_results['total_time'],␣
 ↪'mo-')
        ax4.set_title('Total Time Growth')
        ax4.set_xlabel('Number of Users')
        ax4.set_ylabel('Total Time (seconds)')
        ax4.grid(True)

        plt.tight_layout()
        plt.show()

def main():
    # Create analysis instance
    analyzer = ScalabilityAnalysis()

    # Define test scales
    user_scales = [8, 16, 32, 64, 128]

    # Run analysis
    results = analyzer.run_scalability_analysis(user_scales)

    # Display results table
    print("\nPerformance Analysis Results:")
    print(results.to_string())

    # Plot results
```

```python
    analyzer.plot_scalability_results(results)

if __name__ == "__main__":
    main()
```

```
Analyzing user count: 8
```

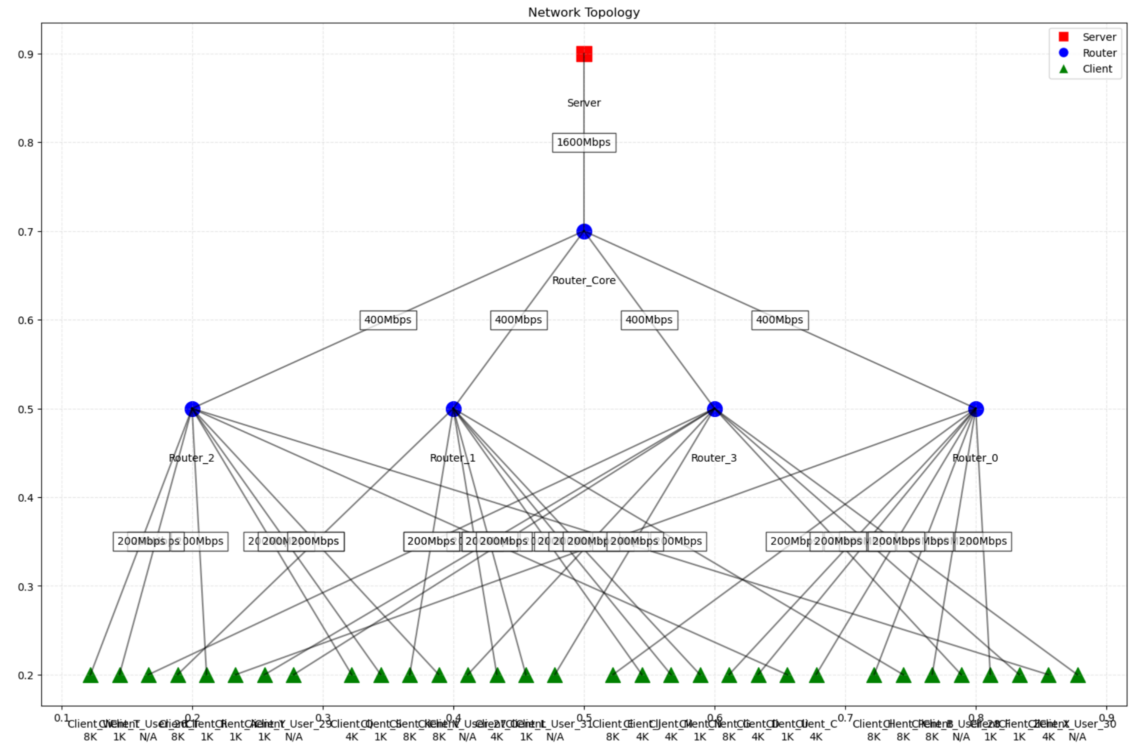



```
Solution status: Success
Completed - Total time: 0.00s

Analyzing user count: 16
```



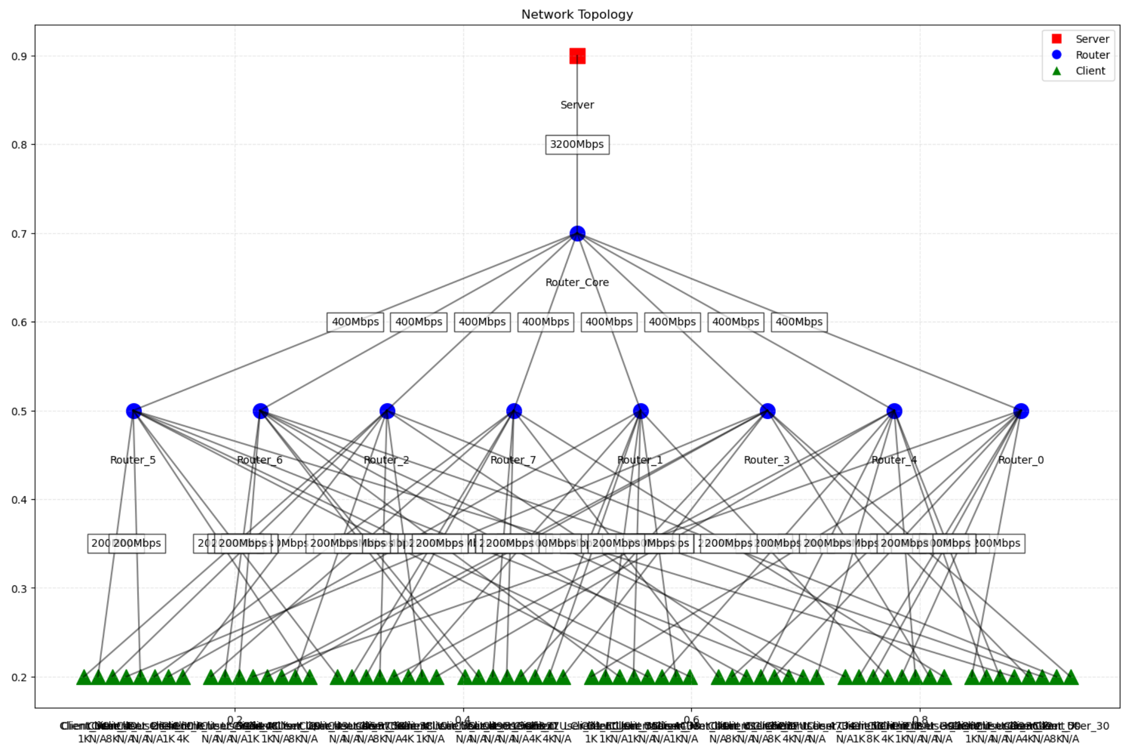
Solution status: Success
Completed - Total time: 0.00s

Analyzing user count: 32

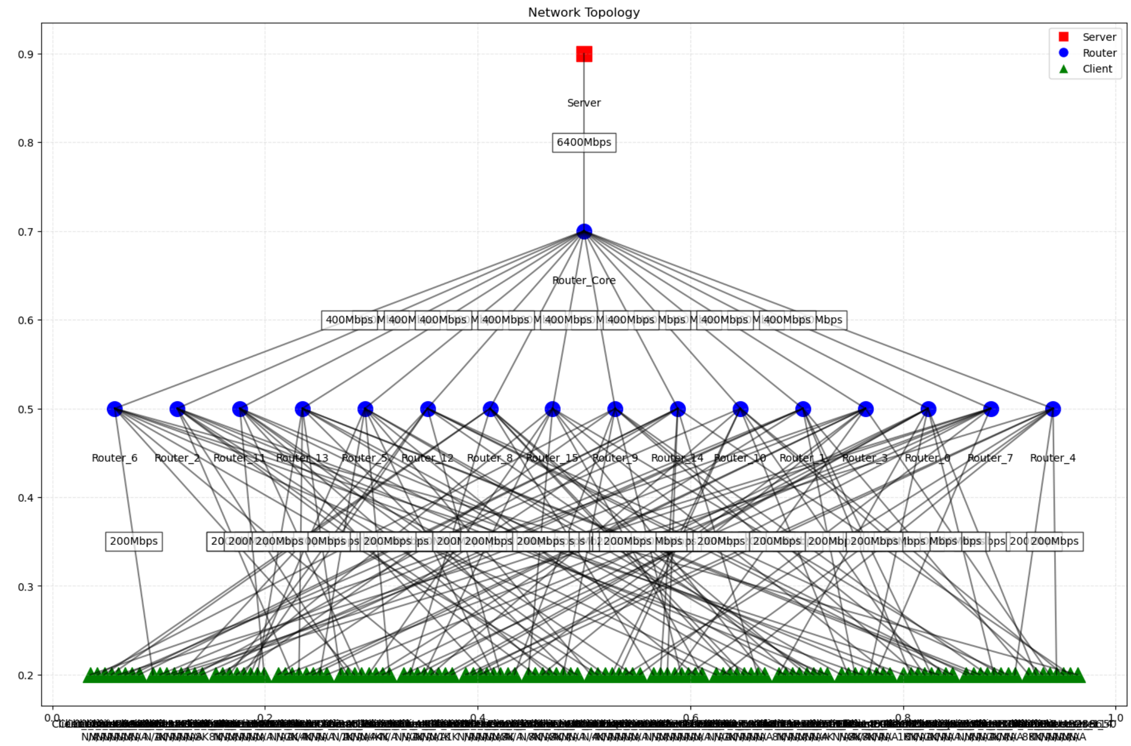


Solution status: Success
Completed - Total time: 0.01s

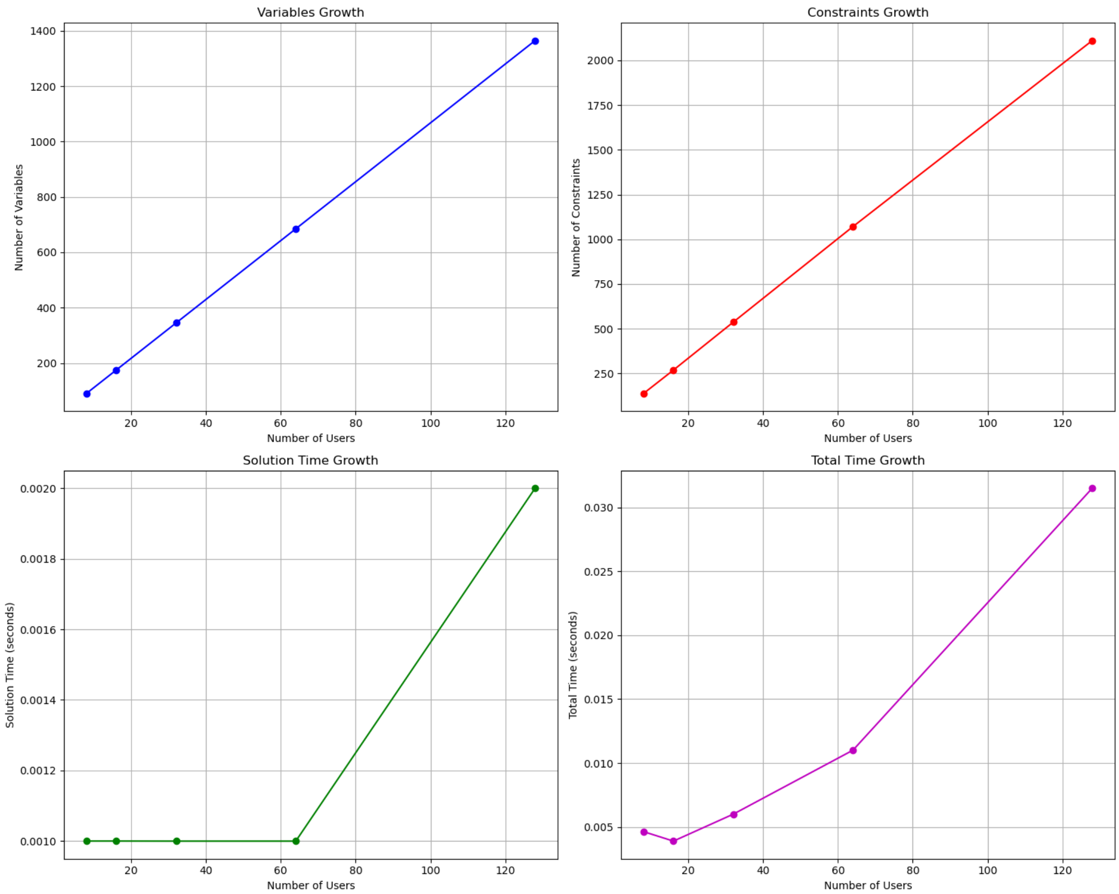
Analyzing user count: 64




```
Solution status: Success
Completed - Total time: 0.01s

Analyzing user count: 128
```




```
Solution status: Success
Completed - Total time: 0.03s

Performance Analysis Results:
   num_users  num_variables  num_constraints  setup_time  solve_time  total_time
objective_value  model_status   status
0          8             90              138    0.003630       0.001    0.004630
-22.0              2  Success
1         16            175              269    0.002888       0.001    0.003888
-49.0              2  Success
2         32            345              537    0.004997       0.001    0.005997
-97.0              2  Success
3         64            685             1071    0.010005       0.001    0.011005
-195.0              2  Success
4        128           1365             2111    0.029518       0.002    0.031518
-419.0              2  Success
```



4 Directly Using Optimization Algorithms (no gorubi)

[6]:
```
import random
import math
import copy
import time
from typing import Dict, List, Tuple
import numpy as np

def measure_execution_time(func):
    """Decorator to measure execution time of methods"""
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Execution time of {func.__name__}: {execution_time:.4f}␣
↪seconds")
```

```
        return result
    return wrapper

class BaseOptimization:
    def __init__(self):
        self.B, self.q, self.max_bitrate, self.links, self.paths = self.
 ↪define_problem_data()
        self.clients = list(self.max_bitrate.keys())
        self.resolutions = list(self.B.keys())

    def define_problem_data(self):
        # Resolution bandwidth requirements (Mbps)
        B = {
            '8K': 200,
            '4K': 45,
            '2K': 16,
            '1K': 8
        }

        # Quality scores
        q = {
            '8K': 4,
            '4K': 3,
            '2K': 2,
            '1K': 1
        }

        # Client maximum bitrates
        max_bitrate = {
            'A': 10,
            'B': 50,
            'C': 200,
            'D': 200,
            'E': 20,
            'F': 50,
            'G': 200,
            'H': 200
        }

        # Network link capacities
        links = {
            ('Server', 'Router_C'): 300,
            ('Router_C', 'Router_A'): 250,
            ('Router_C', 'Router_B'): 250,
            ('Router_A', 'Client_A'): 200,
            ('Router_A', 'Client_B'): 200,
            ('Router_A', 'Client_C'): 200,
```

```
            ('Router_A', 'Client_D'): 200,
            ('Router_B', 'Client_E'): 200,
            ('Router_B', 'Client_F'): 200,
            ('Router_B', 'Client_G'): 200,
            ('Router_B', 'Client_H'): 200
        }

        # Network paths for each client
        paths = {
            'A': [('Server', 'Router_C'), ('Router_C', 'Router_A'), ('Router_A', 'Client_A')],
            'B': [('Server', 'Router_C'), ('Router_C', 'Router_A'), ('Router_A', 'Client_B')],
            'C': [('Server', 'Router_C'), ('Router_C', 'Router_A'), ('Router_A', 'Client_C')],
            'D': [('Server', 'Router_C'), ('Router_C', 'Router_A'), ('Router_A', 'Client_D')],
            'E': [('Server', 'Router_C'), ('Router_C', 'Router_B'), ('Router_B', 'Client_E')],
            'F': [('Server', 'Router_C'), ('Router_C', 'Router_B'), ('Router_B', 'Client_F')],
            'G': [('Server', 'Router_C'), ('Router_C', 'Router_B'), ('Router_B', 'Client_G')],
            'H': [('Server', 'Router_C'), ('Router_C', 'Router_B'), ('Router_B', 'Client_H')]
        }

        return B, q, max_bitrate, links, paths

    def check_capacity_constraints(self, solution: Dict[str, str]) -> bool:
        """Check link capacity constraints"""
        link_usage = {link: 0 for link in self.links.keys()}

        for client, resolution in solution.items():
            bandwidth = self.B[resolution]
            for link in self.paths[client]:
                link_usage[link] += bandwidth

                if link_usage[link] > self.links[link]:
                    return False
        return True

    def calculate_objective(self, solution: Dict[str, str], alpha=1.0, gamma=0.5) -> float:
        """Calculate objective function value"""
        quality_sum = sum(self.q[res] for res in solution.values())
```

```python
        resolution_count = len(solution)

        return -alpha * quality_sum - gamma * resolution_count

class GreedyOptimization(BaseOptimization):
    @measure_execution_time
    def optimize(self) -> Dict[str, str]:
        """
        Greedy algorithm implementation
        Time Complexity Analysis:
        - Sorting clients: O(C log C), where C is number of clients
        - For each client, we try each resolution: O(C * R), where R is number␣
↪of resolutions
        - Checking constraints for each attempt: O(L), where L is number of␣
↪links
        Total Time Complexity: O(C * R * L)
        """
        print("Running Greedy Algorithm:")
        print(f"Number of clients: {len(self.clients)}")
        print(f"Number of resolutions: {len(self.resolutions)}")
        print(f"Number of links: {len(self.links)}")

        solution = {}
        operations_count = 0  # Counter for actual operations

        sorted_clients = sorted(self.clients,
                              key=lambda x: self.max_bitrate[x],
                              reverse=True)

        for client in sorted_clients:
            for resolution in sorted(self.resolutions,
                                  key=lambda x: self.q[x],
                                  reverse=True):
                operations_count += 1
                if self.B[resolution] <= self.max_bitrate[client]:
                    solution[client] = resolution
                    if self.check_capacity_constraints(solution):
                        continue
                    else:
                        solution.pop(client)

            if client not in solution:
                solution[client] = '1K'

        print(f"Total operations performed: {operations_count}")
        return solution
```

```python
class SimulatedAnnealing(BaseOptimization):
    def get_neighbor(self, solution: Dict[str, str]) -> Dict[str, str]:
        """
        Generate a neighbor solution
        Time Complexity: O(1) - Constant time operation
        """
        new_solution = solution.copy()
        client = random.choice(list(solution.keys()))
        current_res = solution[client]

        # Get available resolution options
        available_res = [res for res in self.resolutions
                        if self.B[res] <= self.max_bitrate[client]
                        and res != current_res]

        if available_res:
            new_solution[client] = random.choice(available_res)

        return new_solution

    @measure_execution_time
    def optimize(self, initial_temp=100.0, cooling_rate=0.95,
                iterations=1000) -> Dict[str, str]:
        """
        Simulated Annealing implementation
        Time Complexity Analysis:
        - Initial solution (Greedy): O(C * R * L)
        - For each iteration:
          - Generate neighbor: O(1)
          - Check constraints: O(L)
        Total Time Complexity: O(C * R * L + I * L), where I is number of␣
 ↪iterations
        """
        print("Running Simulated Annealing:")
        print(f"Number of iterations: {iterations}")
        print(f"Initial temperature: {initial_temp}")
        print(f"Cooling rate: {cooling_rate}")

        operations_count = 0
        current_solution = GreedyOptimization().optimize()
        best_solution = current_solution.copy()
        current_cost = self.calculate_objective(current_solution)
        best_cost = current_cost

        temperature = initial_temp

        for i in range(iterations):
```

```
            operations_count += 1
            neighbor = self.get_neighbor(current_solution)

            if self.check_capacity_constraints(neighbor):
                neighbor_cost = self.calculate_objective(neighbor)
                cost_diff = neighbor_cost - current_cost

                if (cost_diff < 0 or
                    random.random() < math.exp(-cost_diff / temperature)):
                    current_solution = neighbor
                    current_cost = neighbor_cost

                    if current_cost < best_cost:
                        best_solution = current_solution.copy()
                        best_cost = current_cost

            temperature *= cooling_rate

            if i % 100 == 0:  # Progress tracking
                print(f"Iteration {i}, Temperature: {temperature:.2f}, Best␣
↪cost: {best_cost:.2f}")

        print(f"Total operations performed: {operations_count}")
        return best_solution

class GeneticAlgorithm(BaseOptimization):
    def create_individual(self) -> Dict[str, str]:
        """Create an individual (solution)"""
        individual = {}
        for client in self.clients:
            available_res = [res for res in self.resolutions
                            if self.B[res] <= self.max_bitrate[client]]
            individual[client] = random.choice(available_res)
        return individual

    def crossover(self, parent1: Dict[str, str],
                  parent2: Dict[str, str]) -> Dict[str, str]:
        """Perform crossover operation"""
        child = {}
        for client in self.clients:
            if random.random() < 0.5:
                child[client] = parent1[client]
            else:
                child[client] = parent2[client]
        return child

    def mutate(self, individual: Dict[str, str],
```

```python
                mutation_rate: float = 0.1) -> Dict[str, str]:
        """Perform mutation operation"""
        mutated = individual.copy()
        for client in self.clients:
            if random.random() < mutation_rate:
                available_res = [res for res in self.resolutions
                               if self.B[res] <= self.max_bitrate[client]]
                mutated[client] = random.choice(available_res)
        return mutated

    @measure_execution_time
    def optimize(self, population_size=50, generations=100) -> Dict[str, str]:
        """
        Genetic Algorithm implementation
        Time Complexity Analysis:
        - Population initialization: O(P * C), where P is population size
        - For each generation:
          - Fitness calculation: O(P * L)
          - Sorting: O(P log P)
          - Creating new population: O(P * C)
        Total Time Complexity: O(G * P * (L + log P + C)), where G is number of␣
↪generations
        """
        print("Running Genetic Algorithm:")
        print(f"Population size: {population_size}")
        print(f"Number of generations: {generations}")

        operations_count = 0
        population = []
        for _ in range(population_size):
            individual = self.create_individual()
            if self.check_capacity_constraints(individual):
                population.append(individual)

        best_solution = None
        best_fitness = float('inf')

        for gen in range(generations):
            operations_count += population_size

            fitness_scores = [(self.calculate_objective(ind), ind)
                            for ind in population]
            fitness_scores.sort(key=lambda x: x[0])

            if fitness_scores[0][0] < best_fitness:
                best_fitness = fitness_scores[0][0]
                best_solution = fitness_scores[0][1].copy()
```

```
            new_population = [ind for _, ind in fitness_scores[:2]]

            while len(new_population) < population_size:
                parent_candidates = fitness_scores[:10]
                parent1 = random.choice(parent_candidates)[1]
                parent2 = random.choice(parent_candidates)[1]

                child = self.crossover(parent1, parent2)
                child = self.mutate(child)

                if self.check_capacity_constraints(child):
                    new_population.append(child)

            population = new_population

            if gen % 10 == 0:  # Progress tracking
                print(f"Generation {gen}, Best fitness: {best_fitness:.2f}")

        print(f"Total operations performed: {operations_count}")
        return best_solution

def compare_methods():
    """Compare the results and performance of different methods"""
    methods = [
        ("Greedy", GreedyOptimization()),
        ("Simulated Annealing", SimulatedAnnealing()),
        ("Genetic Algorithm", GeneticAlgorithm())
    ]

    results = {}
    print("\nPerformance Comparison:")
    print("-" * 50)

    for name, method in methods:
        print(f"\nExecuting {name} Algorithm...")
        start_time = time.time()
        solution = method.optimize()
        end_time = time.time()
        execution_time = end_time - start_time

        objective_value = method.calculate_objective(solution)
        results[name] = {
            'solution': solution,
            'objective_value': objective_value,
            'execution_time': execution_time
        }
```

```
        print(f"\n{name} Results:")
        print(f"Objective Value: {objective_value}")
        print(f"Execution Time: {execution_time:.4f} seconds")
        print("Client Assignments:")
        for client, resolution in solution.items():
            print(f"Client {client}: {resolution}")

    return results

if __name__ == "__main__":
    results = compare_methods()

    # Print comparative summary
    print("\nAlgorithm Comparison Summary:")
    print("-" * 50)
    for name, data in results.items():
        print(f"\n{name}:")
        print(f"Objective Value: {data['objective_value']:.2f}")
        print(f"Execution Time: {data['execution_time']:.4f} seconds")
```

```
Performance Comparison:
--------------------------------------------------

Executing Greedy Algorithm…
Running Greedy Algorithm:
Number of clients: 8
Number of resolutions: 4
Number of links: 11
Total operations performed: 32
Execution time of optimize: 0.0012 seconds

Greedy Results:
Objective Value: -12.0
Execution Time: 0.0012 seconds
Client Assignments:
Client C: 1K
Client D: 1K
Client G: 1K
Client H: 1K
Client B: 1K
Client F: 1K
Client E: 1K
Client A: 1K

Executing Simulated Annealing Algorithm…
Running Simulated Annealing:
```

```
Number of iterations: 1000
Initial temperature: 100.0
Cooling rate: 0.95
Running Greedy Algorithm:
Number of clients: 8
Number of resolutions: 4
Number of links: 11
Total operations performed: 32
Execution time of optimize: 0.0000 seconds
Iteration 0, Temperature: 95.00, Best cost: -13.00
Iteration 100, Temperature: 0.56, Best cost: -23.00
Iteration 200, Temperature: 0.00, Best cost: -25.00
Iteration 300, Temperature: 0.00, Best cost: -25.00
Iteration 400, Temperature: 0.00, Best cost: -25.00
Iteration 500, Temperature: 0.00, Best cost: -25.00
Iteration 600, Temperature: 0.00, Best cost: -25.00
Iteration 700, Temperature: 0.00, Best cost: -25.00
Iteration 800, Temperature: 0.00, Best cost: -25.00
Iteration 900, Temperature: 0.00, Best cost: -25.00
Total operations performed: 1000
Execution time of optimize: 0.0040 seconds

Simulated Annealing Results:
Objective Value: -25.0
Execution Time: 0.0040 seconds
Client Assignments:
Client C: 4K
Client D: 4K
Client G: 4K
Client H: 4K
Client B: 4K
Client F: 4K
Client E: 2K
Client A: 1K

Executing Genetic Algorithm Algorithm…
Running Genetic Algorithm:
Population size: 50
Number of generations: 100
Generation 0, Best fitness: -23.00
Generation 10, Best fitness: -25.00
Generation 20, Best fitness: -25.00
Generation 30, Best fitness: -25.00
Generation 40, Best fitness: -25.00
Generation 50, Best fitness: -25.00
Generation 60, Best fitness: -25.00
Generation 70, Best fitness: -25.00
Generation 80, Best fitness: -25.00
```

```
Generation 90, Best fitness: -25.00
Total operations performed: 5000
Execution time of optimize: 0.0302 seconds

Genetic Algorithm Results:
Objective Value: -25.0
Execution Time: 0.0302 seconds
Client Assignments:
Client A: 1K
Client B: 4K
Client C: 4K
Client D: 4K
Client E: 2K
Client F: 4K
Client G: 4K
Client H: 4K

Algorithm Comparison Summary:
--------------------------------------------------

Greedy:
Objective Value: -12.00
Execution Time: 0.0012 seconds

Simulated Annealing:
Objective Value: -25.00
Execution Time: 0.0040 seconds

Genetic Algorithm:
Objective Value: -25.00
Execution Time: 0.0302 seconds
```

4.1 Analysis of Optimization Algorithms

4.1.1 1. Greedy Algorithm

- **Approach**: Makes locally optimal choices at each step
- **Implementation Logic**:
  - Sorts clients by maximum bandwidth (highest to lowest)
  - For each client, tries to assign highest quality resolution that satisfies constraints
  - Falls back to lowest resolution (1K) if no feasible solution found
- **Advantages**: Fast, simple, deterministic
- **Disadvantages**: May get stuck in local optima

4.1.2 2. Simulated Annealing

- **Approach**: Probabilistic technique that simulates physical annealing process
- **Implementation Logic**:
  - Starts with greedy solution

- Iteratively generates neighbor solutions by randomly changing one client's resolution
  - Accepts improvements always, accepts worse solutions with decreasing probability
  - Uses temperature parameter to control acceptance of worse solutions
- **Advantages**: Can escape local optima, good balance of exploration and exploitation
- **Disadvantages**: Results may vary between runs, requires parameter tuning

4.1.3 3. Genetic Algorithm

- **Implementation Logic**:
  - Population-based approach with selection, crossover, and mutation
  - Uses elitism to preserve best solutions
  - Crossover randomly selects resolution assignments from parents
  - Mutation randomly changes resolutions with low probability
  - Maintains feasibility through constraint checking
- **Advantages**: Can explore multiple solution paths simultaneously, good for complex search spaces
- **Disadvantages**: Computationally intensive, requires careful parameter tuning

4.2 Comparison of Approaches

4.2.1 Performance Characteristics

- **Greedy**: Generally provides good solutions quickly, but may miss global optimum
- **Simulated Annealing**: Often finds better solutions than greedy, but takes longer
- **Genetic Algorithm**: Can find high-quality solutions, but requires most computational resources

4.2.2 Use Case Recommendations

- **Greedy**: Best for quick solutions or as initial solutions for other methods
- **Simulated Annealing**: Good for medium-sized problems where solution quality is important
- **Genetic Algorithm**: Best for complex problems where computational time is not a major constraint