课程设计实验报告

Project1: 网络流量优化问题

姓 名: 王 瑞 祺

学 号: 202421011308

通信网理论

(秋季, 2024)

电子科技大学

信息与通信工程学院

2024 年 11 月 6 日

目 录

1 问题描述

给定一个网络 $G(\mathcal{V},\mathcal{E})$ ，其中 $\mathcal{V}$ 为节点集合，$\mathcal{V}$ 为链路集合。网络中的每条链路 e 的容量为 C_e 拓扑上的数字为链路的容量，假设网络中有 K 条单向网络流（$k=n*(n-1)$,n 为网络节点的数目），假定第 i 条网络流为 f_i，流的大小从 $[10,100]$ 区间中随机产生。现需要对 K 条网络流进行合理的规划，以实现**网络负载均衡**的目标。假定网络负载均衡的指标为**最小化最大链路利用率**。完成下面两个任务：

(1) 采用 Link-Path 或者 Node-Link 的方式进行建模，并使用求解器（CPLEX，Matlab，Python）对问题进行求解（如果采用 Link-path 进行建模，要求为每条流使用 K 路由算法计算 3 条备选路）。

(2) 推导（1）中建立模型的对偶模型，并采用求解器求解对偶模型。

(3) 比较原来模型和对偶模型的求解时间和优化目标值，并分析结论。

建议：使用 CPLEX 工具进行建模和求解。

2 模型建立

此处采用 Link-Path 方式对该网路流量优化问题进行建模。

2.1 符号定义

区分该问题中的常数与决策变量对于线性规划的构建非常重要，因此此处将分别介绍常数符号和决策变量符号以避免混淆。

2.1.1 常数符号

物理含义	数学符号
所有节点的集合, 所有节点的集合，所有流的集合，流的所有可选路径的集合	$\mathcal{V},\mathcal{E},\mathcal{F},\mathcal{P}$
$\mathcal{P}$ 第 i 条流量, $\mathcal{E}$ 中的一条链路	f_i, e
f_i 可选路径的集合, 链路 e 的容量	$\mathcal{P}_i, C_e$
f_i 的第 j 条可用路径	$p_{i,j}$
表示 $p_{i,j}$ 是否使用链路 e 的二进制常量	$\delta_{i,j}^e$

2.1.2 决策变量符号

需要对网络流进行合理的规划以实现最小化最大链路利用率，因此可以将该问题的决策变量定义为:

- **路径流量百分比变量** $\alpha_{i,j}$: f_i 在 $p_{i,j}$ 上的流量百分比
- **链路使用率变量** u_e: 表示每条链路 e 的利用率，即该链路上的负载与其容量的比值。

2.2 目标函数

目标是最小化所有链路的最大利用率，因此目标函数可写为:

$$\min \max_{e\in\mathcal{E}} u_e, \tag{1}$$

引入变量 U 来表示最大链路利用率，使得目标函数变为:

$$\min U, \tag{2}$$

对于每一条链路 e 都应该满足约束 $u_e \leq U$ 以保证 U 是最大链路使用率。

2.3 约束

2.3.1 流量守恒约束

对于每条流量 f_i, 其可用路径上的百分比流量之和应等于该流量，即该流量的路径流量百分比变量之和应等于 1，可表示为:

$$\sum_{p_{i,j}\in\mathcal{P}_i} \alpha_{i,j} = 1, \forall f_i \in \mathcal{F}, \tag{3}$$

2.3.2　链路容量约束

对于每条链路 e，其上所有需求所产生的总流量不能超过其链路容量，可表示为：

$$\sum_{f_i \in \mathcal{F}} \sum_{p_{i,j} \in \mathcal{P}_i} \delta_{i,j}^e \cdot \alpha_{i,j} \cdot f_i \leq C_e, \forall e \in \mathcal{E}, \tag{4}$$

2.3.3　最大链路使用率约束

每条链路 e 的链路使用率都不能大于最大链路使用率 U, 根据公式 (7), 每条链路 e 的链路使用率 u_e 可写为:

$$C_e = \frac{\sum_{f_i \in \mathcal{F}} \sum_{p_{i,j} \in \mathcal{P}_i} \delta_{i,j}^e \cdot \alpha_{i,j} \cdot f_i}{C_e}, \forall e \in \mathcal{E}, \tag{5}$$

因此，最大链路使用率约束可表示为：

$$\frac{\sum_{f_i \in \mathcal{F}} \sum_{p_{i,j} \in \mathcal{P}_i} \delta_{i,j}^e \cdot \alpha_{i,j} \cdot f_i}{C_e} - U \leq 0, \forall e \in \mathcal{E}, \tag{6}$$

公式 (4) 和公式 (6) 可写为同一个约束，由以下公式表示：

$$\sum_{f_i \in \mathcal{F}} \sum_{p_{i,j} \in \mathcal{P}_i} \delta_{i,j}^e \cdot \alpha_{i,j} \cdot f_i \leq U \cdot C_e, \forall e \in \mathcal{E}, \tag{7}$$

2.3.4　变量约束

所有的流量百分比变量都必须是非负的，即：

$$\alpha_{i,j} \geq 0, \forall p_{i.j} \in \mathcal{P}_i, \forall f_i \in \mathcal{F} \tag{8}$$

2.4　对偶模型

根据以上的分析，原问题经过 Link-Path 的方式建模后得到的线性规划问题可写作：

$$\begin{aligned} \min \quad & U \\ \text{s.t.} \quad & \sum_{p_{i,j} \in \mathcal{P}_i} \alpha_{i,j} = 1, \forall f_i \in \mathcal{F}, \\ & \sum_{f_i \in \mathcal{F}} \sum_{p_{i,j} \in \mathcal{P}_i} \delta_{i,j}^e \cdot \alpha_{i,j} \cdot f_i \leq U \cdot C_e, \forall e \in \mathcal{E}, \\ & \alpha_{i,j} \geq 0, \forall p_{i.j} \in \mathcal{P}_i, \forall f_i \in \mathcal{F}, \end{aligned} \tag{9}$$

利用拉格朗日乘子法求该线性规划的对偶，为每个约束项引入拉格朗日乘子，构造拉格朗日函数：

$$\begin{aligned} L(\alpha, U, m, n) = U &+ \sum_{f_i \in \mathcal{F}} (1 - \sum_{p_{i,j} \in P_i} \alpha_{i,j}) m_i \\ &+ \sum_{e \in \mathcal{E}} (\sum_{f_i \in \mathcal{F}} \sum_{p_{i,j} \in \mathcal{P}_i} \delta_{i,j}^e \cdot \alpha_{i,j} \cdot f_i - U \cdot C_e) n_e \end{aligned} \tag{10}$$

提取出原线性规划的决策变量得到：

$$
\begin{aligned}
L(\alpha, U, m, n) = & \sum_{f_i \in \mathcal{F}} m_i + (1 - \sum_{e \in \mathcal{E}} C_e \cdot n_e) U \\
& + \sum_{f_i \in \mathcal{F}} \sum_{p_{i,j} \in \mathcal{P}_i} (\sum_{e \in \mathcal{E}} \delta_{i,j}^e \cdot f_i \cdot n_e - m_i) \alpha_{i,j}
\end{aligned} \tag{11}
$$

根据原线性规划各约束，对偶线性规划可写为：

$$
\begin{aligned}
\max \quad & U \\
\text{s.t.} \quad & 1 - \sum_{e \in \mathcal{E}} C_e \cdot n_e = 0, \\
& \sum_{e \in \mathcal{E}} \delta_{i,j}^e \cdot f_i \cdot n_e - m_i \geq 0, \forall p_{i.j} \in \mathcal{P}_i, \forall f_i \in \mathcal{F} \\
& n_e \geq 0, \forall e \in \mathcal{E}.
\end{aligned} \tag{12}
$$

3 模型实现

3.1 网络环境 (Network_Topology.py)

网络拓扑的邻接矩阵：

```
self.graph = {
            "Copenhagen": {"London": 1, "Amsterdam": 1, "Berlin":1},
            "London": {"Copenhagen": 1, "Amsterdam": 1, "Brussels": 1,
             ↪  "Paris":1},
            "Amsterdam": {"Copenhagen": 1, "London": 1, "Brussels": 1,
             ↪  "Luxemburg": 1, "Berlin": 1},
            "Berlin": {"Copenhagen": 1, "Amsterdam": 1, "Paris": 1, "Prague":
             ↪  1, "Vienna":1},
            "Brussels": {"London": 1, "Amsterdam": 1, "Luxemburg": 1,
             ↪  "Paris":1},
            "Luxemburg": {"Amsterdam": 1, "Prague": 1, "Zurich": 1, "Paris":1,
             ↪  "Brussels":1},
            "Prague": {"Copenhagen": 1, "Berlin": 1, "Vienna": 1, "Zurich": 1,
             ↪  "Luxemburg": 1},
            "Paris": {"London": 1, "Brussels": 1, "Luxemburg": 1, "Berlin":1,
             ↪  "Zurich":1, "Milan":1},
            "Zurich": {"Luxemburg": 1, "Prague": 1, "Vienna": 1, "Milan": 1,
             ↪  "Milan": 1},
            "Vienna": {"Berlin": 1, "Prague": 1, "Zurich": 1, "Milan": 1},
            "Milan": {"Paris": 1, "Brussels": 1, "Zurich": 1, "Vienna": 1},
        }
```

在 [1,5] 区间随机生成链路权重：

```
 for node in self.graph:
            for adj_node in self.graph[node]:
                self.graph[node][adj_node] = random.randint(1, 5)
                self.graph[adj_node][node] = self.graph[node][adj_node]
```

在 [200,1000] 区间随机生成链路容量：

```
self.link_capacity = copy.deepcopy(self.graph)
for node in self.graph:
    for adj_node in self.graph[node]:
        self.link_capacity[node][adj_node] = random.randint(20, 100)*10
        self.link_capacity[adj_node][node] =
         ↪  self.link_capacity[node][adj_node]
```

生成 110 条流量，流量大小从 [10,100] 的区间随机生成：

```
self.flow = dict()
        for src in self.graph.keys():
            self.flow[src] = dict()
            for des in self.graph.keys():
                if des != src:
                    self.flow[src][des] = random.randint(10, 100)
```

3.2 K 路由算法 (k_shortest_path.py)

dijkstra 函数实现 dijstra 算法，得到从起始点 (src) 到目标点 (des) 的最短路径：

```
def dijkstra(gp: net_graph, src, des, node_set: set = None):
    distances = dict()
    parent = dict()
    visited = set()
    path = list()

    for node in gp.graph:
        distances[node] = gp.INFINITY # 初始时，所有节点到达初始节点的距离设置为
         ↪  无穷大
        parent[node] = None

    if node_set is not None:
        visited = copy.deepcopy(node_set)
    visited.add(src)
```

```python

    distances[src] = 0 # 起始点与起始点的距离为 0
    # 遍历起始点的相邻点
    for node_adj in gp.graph[src]:
        parent[node_adj] = src
        # 起始点与相邻点的距离等于链路权重
        distances[node_adj] = gp.graph[src][node_adj]

    node_min_weight = gp.INFINITY
    node_min = src # 当前正在处理的节点

    while True:
        for node in distances:
            # 找到距离节点最近的相邻点
            if node not in visited and distances[node] < node_min_weight:
                node_min = node
                node_min_weight = distances[node]

        if node_min == des:
            path, path_weight = build_path(gp, parent, src, des)
            return path, path_weight
            path_weight = 0
            # 通过 parent 字典反向构建最短路径
            while node_min != src:
                path.append(node_min)
                path_weight += gp.graph[node_min][parent[node_min]]
                node_min = parent[node_min]
            path.append(src)
            # 列表翻转，得到最短路径 path
            path.reverse()
            # 返回包含构成最短路径的节点列表和权重和
            return path, path_weight

        # 跳过已经处理过的节点避免重复处理
        if node_min in visited:
            break

        visited.add(node_min)
        for node_adj in gp.graph[node_min]:
            if distances[node_min] + gp.graph[node_min][node_adj] < distances[node_adj]:
                parent[node_adj] = node_min
                # 更新当前节点与起始节点的距离
```

```
                distances[node_adj] = distances[node_min] + gp.graph[node_min][node_adj]

        node_min_weight = gp.INFINITY

    return [], 0
```

ksp 函数在函数使用了 Dijkstra 算法的基础上，寻找源节点 (src) 到目标节点 (des) 的前 k 条最短路径:

```
def ksp(gp: net_graph, src, des, max_k):
    if max_k < 1:
        return {}

    paths = dict()
    paths_set = set()
    paths_weight = dict()
    paths_deviate = list()

    # 通过 Dijkstra 算法找到最短路径
    path_tmp, path_weight_tmp = dijkstra(gp, src, des)

    hq.heappush(paths_deviate, (path_weight_tmp, path_tmp))
    paths_set.add(tuple(path_tmp))

    num = 0
    while paths_deviate:
        paths_weight[num], paths[num] = hq.heappop(paths_deviate)
        paths_set.remove(tuple(paths[num]))
        path_now = paths[num]
        num += 1

        if num >= max_k:
            break

        for index in range(len(path_now) - 1):
            gp_tmp = copy.deepcopy(gp)
            remove_edges(gp_tmp, paths, path_now, index)
            for path in paths:
                if paths[path][:index + 1] == path_now[:index + 1]:
                    each_path = paths[path]
                    if each_path[index + 1] in gp_tmp.graph[each_path[index]]:
```

```python
                        del gp_tmp.graph[each_path[index]][each_path[index + 1]]
                        del gp_tmp.graph[each_path[index + 1]][each_path[index]]

            path_tmp, path_weight_tmp = dijkstra(gp_tmp, path_now[index], des, set(path_now[:index + 1]))
            if path_weight_tmp == 0:
                continue
            path_s2t_tmp = path_now[:index + 1] + path_tmp[1:]

            if tuple(path_s2t_tmp) not in paths_set:
                hq.heappush(paths_deviate, (path_weight_tmp, path_s2t_tmp))
                paths_set.add(tuple(path_s2t_tmp))

    return paths
```

3.3 线性规划求解 (solver.py)

添加路径流量百分比变量 $\alpha_{i,j}$:

```python
for src in net.flow:
        for des in net.flow[src]:
            paths[(src, des)] = ksp(net, src, des, max_k)
            for i in range(max_k):
                name = "source:{}_destination:{}_p{}".format(src, des, i + 1)
                alpha[name] = cplex_obj.variables.add(names=[name], types=['C'], lb=[0], ub=[1])
```

添加最大链路利用率变量 U:

```python
utility = cplex_obj.variables.add(names=["utility"], types=['C'], lb=[0], ub=[1])
```

添加如公式 (3) 所示的流量守恒约束:

```python
for node in net.link_capacity:
    for node_adj in net.link_capacity[node]:
        lin_expr_vars = ["source:{}_destination:{}_p{}".format(src, dst, i + 1) for i in range(max_k)]
            lin_expr_coeffs = [1.0 for i in range(max_k)]
```

```
            cplex_obj.linear_constraints.add(lin_expr=[[lin_expr_vars, lin_expr_coeffs]], senses=['E'], rhs=[1])
```

添加如公式 (7) 所示的链路约束：

```
for node in net.link_capacity:
    for node_adj in net.link_capacity[node]:
        lin_expr_vars = [name for name in alpha.keys()]
        lin_expr_coeffs = []
        lin_expr_coeffs_2 = []
        for src in net.flow:
            for des in net.flow[src]:
                for i in range(max_k):
                    lin_expr_coeffs = lin_expr_coeffs + [delta[(node, node_adj)][(src, des, i)]*net.flow[src][des]]
                    lin_expr_coeffs_2 = lin_expr_coeffs_2 + [delta[(node, node_adj)][(src, des, i)]*net.flow[src][des]/
                    net.link_capacity[node][node_adj]]
        cplex_obj.linear_constraints.add(lin_expr=[[lin_expr_vars, lin_expr_coeffs]], senses=['L'], rhs=[net.link_capacity[node][node_adj]])
        cplex_obj.linear_constraints.add(lin_expr=[[lin_expr_vars+['utility'], lin_expr_coeffs_2 +[-1]]], senses=['L'], rhs=[0])
```

添加如公式 (2) 的目标函数，设置优化方向，进行求解：

```
cplex_obj.objective.set_linear([('utility', 1.0)])
cplex_obj.objective.set_sense(cplex_obj.objective.sense.minimize)
cplex_obj.solve()
```

3.4 对偶线性规划求解 (solver_dual.py)

添加变量 n：

```
n = dict()
n_name = []
for node in net.link_capacity:
    for node_adj in net.link_capacity[node]:
        name = "node:{}_node_adj:{}".format(node, node_adj)
        n_name.append(name)
        n[name] = cplex_obj.variables.add(names=[name], types=['C'], lb=[0])
```

添加变量 m：

```
m = dict()
m_name = []
for src in net.flow:
    for des in net.flow[src]:
        paths[(src, des)] = ksp(net, src, des, max_k)
        name = "source:{}_destination:{}".format(src, des)
        m_name.append(name)
        m[name] = cplex_obj.variables.add(names=[name], types=['C'])
```

添加约束 1：

```
lin_expr_vars_1 = [name for name in n_name]
lin_expr_coeffs_1 = []
for node in net.link_capacity:
    for node_adj in net.link_capacity[node]:
        lin_expr_coeffs_1 = lin_expr_coeffs_1 + [net.link_capacity[node][node_adj]]
cplex_obj.linear_constraints.add(lin_expr=[[lin_expr_vars_1, lin_expr_coeffs_1]],senses=['E'], rhs=[1])
```

添加约束 2：

```
for src in net.flow:
    for des in net.flow[src]:
        for i in range(max_k):
            lin_expr_vars_2 = [f"source:{src}_destination:{des}"] + n_name
            lin_expr_coeffs_2 = [-1]
            for node in net.link_capacity:
                for node_adj in net.link_capacity[node]:
                    lin_expr_coeffs_2 = lin_expr_coeffs_2 + [delta[(node, node_adj)][(src, des, i)] * net.flow[src][des]]
            cplex_obj.linear_constraints.add(lin_expr=[[lin_expr_vars_2, lin_expr_coeffs_2]],senses=['G'], rhs=[0])
```

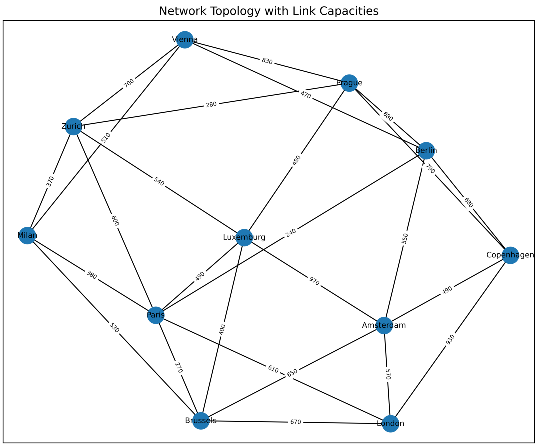
添加目标函数，设置优化方向，进行求解：

```
cplex_obj.objective.set_linear([(name, 1.0) for name in m_name])
cplex_obj.objective.set_sense(cplex_obj.objective.sense.maximize)
cplex_obj.solve()
```

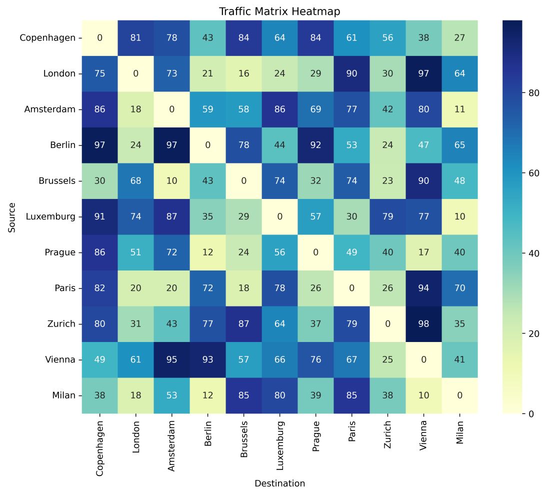
4 结果及分析

4.1 实验设置

实验是在一个节点数量为 11，链路数为 52 的网络中进行。网络中链路的容量属于 [100,1000] 区间，流的大小从 [10,100] 区间中随机产生。网络拓扑及链路容量如图1a所示，流量矩阵的热力图如图1b所示。



(a) 网络拓扑



(b) 流量矩阵的热力图

图 1: 仿真环境设置

4.2 结果分析

在每条流可选用的路径数量 $k = 3, 5, 7$ 时分别进行了三组实验，实验结果如下表所示：

可选路径数 k	原线性规划		对偶线性规划	
	优化目标值	求解时间 (s)	优化目标值	求解时间 (s)
3	0.852	0.0160	0.852	0.0160
5	0.655	0.0310	0.655	0.0160
7	0.592	0.0160	0.592	0.0150

表 1: 原线性规划与对偶线性规划的优化目标值和求解时间对比

从表格中可以看出，原线性规划和对偶线性规划的优化目标值在各个可选路径数 $k = 3, 5, 7$ 下是一致的。这种结果符合线性规划的**强对偶定理**，即在最优解下，原问题和对偶问题的目标值应当相等。因此，从优化目标值上来看，原线性规划和对偶线性规划在结果上是等价的。

在求解时间上，原线性规划和对偶线性规划存在一些细微差异，对偶线性规划在求解时间上比原线性规划略有优势，但总体差异不大。在此问题的求解场景下中，对偶问题在某些情况下求解时间稍短，可能是由于对偶问题的结构使得某些计算过程更为简化。

通过以上的分析可得线性规划及其对偶线性规划的结论：

- **优化目标值一致**：原模型和对偶模型在最优解下的目标值相同，符合强对偶定理。这意味着在优化目标方面，两者的表现是完全等价的。
- **对偶模型对于特定问题可简化求解**：对偶问题在某些场景下可以减少求解器的计算复杂度，特别是在约束条件和决策变量上的简化。