南開大學
Nankai University

计算机学院

算法导论实验报告

多源多汇邮件输送问题

姓名：葛明宇

学号：2312388

专业：计算机科学与技术

2025 年 6 月 10 日

目录

1 背景

在当今社会，物流行业的迅猛发展促使邮件输送的高效性成为衡量物流系统性能的关键指标。邮件物流体系涵盖邮件分拣中心、目的地邮局及邮件处理中心等多个节点，各节点间传输通道的容量限制使得邮件输送策略的制定面临挑战。如何在满足容量限制条件下，实现从源点至汇点的邮件最大输送量，成为一个亟待解决且极具现实意义的问题。本实验旨在深入探究一种有效的邮件输送方案，以期提升物流效率、降低成本，进而为用户提供了一种优化的邮件输送策略。

2 问题描述

在邮件物流系统里，有多个邮件分拣中心作为源点，每个分拣中心单位时间会产生一定数量的邮件，同时，存在多个目的地邮局作为汇点，需要接收相应数量的邮件，不过单位时间能够接收的邮件数量是有上限的。邮件在传输过程中会经过一些邮件处理中心，这些处理中心本身不产生或消耗邮件，仅起到中转的作用。节点之间通过有向路径连接，每条路径都有其容量限制，即单位时间内能够传输的最大邮件数量。目标是设计一种邮件输送方案，使得在满足所有传输通道容量限制的情况下，单位时间内，邮件从源点成功输送至汇点达到最大数目。

3 模型抽象

3.1 图的构建

将邮件物流系统抽象为一个有向图 $G = (V, E)$, 其中 V 是节点集合，包括源点（邮件分拣中心）、汇点（目的地邮局）和中转节点（邮件处理中心）；E 是边集合，代表节点间的有向传输通道。这里设定 G 是弱连通的，不会存在环路。在我们的模型中对原问题做了简化。主要是对于任意一点到另一点的路径大小都是固定的，这保证了在每一时刻，运输时不会产生异步逻辑。即使分拣中心单位时间产生的邮件数量导致了路径选择的变化，也不会对在此之前已经运输的邮件产生影响。否则可能出现不同时刻运输的邮件在某一时刻出现在了同一中转或者汇点处，使我们的运输方案产生了冲突。当然, 如果不考虑异步等情况 (现实中还需要考虑到路程, 而每条路的长度是不确定的), 只需要得到当前状态的最优路径的情况下, 即使任意一点到另一点存在不同路径, 且长度不一，我们的算法也是能够得到最佳方案的。

3.2 节点属性

(1) 源点：具有初始邮件供应量，表示该分拣中心产生的邮件数量。

(2) 汇点：具有邮件接收量，表示该目的地邮局最大接收的邮件数量。

(3) 中转节点：无生产和消耗能力，仅用于邮件中转。

上述的量均以单位时间为标准，后续就不再说明。

3.3 边属性

每条边 $e = (u, v)$ 具有容量 $c(u, v)$, 表示从节点 u 到节点 v 的传输通道单位时间内能够传输的最大邮件数量。

3.4 超级节点

为了方便处理多源多汇问题，引入一个超级源点 s 和一个超级汇点 t。超级源点与所有源点相连，边的容量为对应源点的初始邮件供应量; 所有汇点与超级汇点相连，边的容量为对应汇点的邮件最大容量。

4 算法设计

4.1 算法描述

首先引入超级源点 s 和超级汇点 t ，将多源多汇问题转化成单源单汇问题，不过与单源单汇相比，需要扩展 s 和其他源点间的容量，这里就使用每个源点的初始邮件供应量；以及需要扩展 t 和其他汇点间的容量，这里就使用每个汇点的最大接收容量。

而解决单源单汇问题算法选择采用带缩放的 $Ford-Fulkerson$ 算法。该算法的核心思想是不断寻找增广路径，并沿着增广路径增加流量，直到找不到增广路径为止。带缩放的 $Ford-Fulkerson$ 算法通过引入一个缩放因子 δ，减少了不必要的增广路径搜索，提高了算法的效率。

4.2 伪代码

Algorithm 1 带缩放的多源多汇最大流算法——Scaling Max Flow

Input: 流网络 $G=(V,E)$，源点 s_1、s_2、... ，汇点 t_1、t_2、... ，容量函数 c

Output: 从 s_1、s_2、... 到 t_1、t_2、... 的最大流 f

```
function SCALING-MAX-FLOW(G, s, t, c)
    添加超级源点到源点的边，容量为源点的初始供应
    添加超级汇点到汇点的边，容量为汇点的最大接收容量
    初始化所有边的流量 f(e) ← 0，对所有 e ∈ E
    Δ ← 不大于所有源点初始邮件供应量之和的最大 2 的幂
    G_f ← 初始残余图
    while Δ ≥ 1 do
        G_f(Δ) ← Δ-残余图，仅包含残余容量 ≥ Δ 的边
        while G_f(Δ) 中存在增广路径 P do
            f ← 沿路径 P 更新流量（增加 Δ(P)）
            更新 G_f(Δ)（移除残余容量 < Δ 的边）
        end while
        Δ ← Δ/2
    end while
    return f
end function
```

4.3 复杂度分析

4.3.1 时间复杂度

设定 C 是所有源点初始供应量之和。首先是将多源多汇的模型转为单源单汇的模型。由于将 s 与 t 加入到图中一定不超过原图的 $Edges$ 数, 所以在构建新的图 G_r 时，时间复杂度为 $O(E)$，这里 E 既

可以是原图 G 的边数，也可以是新图 G_r 的边数。

δ 的取值次数：由于 δ 初始值为小于等于 C 的最大的 2 的幂次方，并且每次迭代将 δ 减半，所以 δ 的取值次数为 $O(\log C)$。

每次 δ 下的增广路径查找：在每一个 δ 值下，每次增广至少使流增加 δ。因为每次增广至少要经过一条边，而每条边的剩余容量至少为 δ。在一个 δ 值下，增广的次数最多为 $2E$，每次增广需要 $O(E)$ 的时间来进行深度优先搜索（DFS）查找增广路径。所以，在每一个 δ 值下，增广的总时间复杂度为 $O(E^2)$。

综合以上两点，带缩放的多源多汇 $Ford-Fulkerson$ 算法的总时间复杂度为 $O(E^2 \log C)$。

4.3.2 空间复杂度

该算法主要使用邻接表来存储图的信息，邻接表需要存储每条边的信息，因此空间复杂度为 $O(E)$，其中 E 是图中边的数量。在代码中，$vector < vector < Edge >> adj$ 用于存储邻接表，它的空间开销主要取决于边的数量。

5 算法实现

5.1 代码整体结构

代码主要由以下几个部分组成：

(1) 结构体 $Edge$: 用于表示图中的边，包含边的终点、容量、流量和反向边的索引。

(2) 类 $MaxFlow$: 封装了图的表示和最大流算法的实现，包括添加边、深度优先搜索（DFS）、带缩放的 $Ford-Fulkerson$ 算法以及打印图信息的方法。

(3) 测试函数 $testCase$: 用于构建测试用的图，并调用 $MaxFlow$ 类的方法计算最大流和输出结果。

(4) 主函数 $main$: 定义多个测试用例，调用 $testCase$ 函数进行测试。

5.2 代码详解

结构体 $Edge$

结构体中定义了 4 个成员, 分别是 to 表示边的终点节点;$capacity$ 表示边的最大容量;$flow$ 表示当前边的流量;$reverse$ 表示反向边在邻接表中的索引, 用于更新反向边的流量。由于后续按照邻接表存储边，所以 Edge 中不需要加入 $from$ 成员，表示起点。

```
struct Edge {
    int to;          // 终点
    int capacity;    // 容量
    int flow;        // 流量
    int reverse;     // 反向边索引
};
```

类 $MaxFlow$

成员变量 n 表示图中节点的数量;adj 表示邻接表, 用于存储图的边信息。

```
class MaxFlow {
public:
    int n;                              // 节点数
    vector<vector<Edge>> adj;           // 邻接表
    ...
};
```

使用 *addEdge*() 向图中添加一条有向边，并同时添加其反向边。正向边的容量为 *capacity*, 流量初始化为 0; 反向边的容量初始化为 0, 流量也初始化为 0。

```
class MaxFlow {
public:
    ...
    // 添加边，同时添加反向边
    void addEdge(int from, int to, int capacity) {
        adj[from].push_back({ to, capacity, 0, (int)adj[to].size() });
        adj[to].push_back({ from, 0, 0, (int)adj[from].size() - 1 });
    }
};
```

深度优先搜索方法 *dfs*() 用于在残余网络中寻找一条残余容量不小于 δ 的增广路径。其中,*u* 表示当前节点;*t* 表示汇点;*delta* 表示缩放因子;*visited* 表示标记数组, 用于记录节点是否已访问;*path* 表示存储增广路径上的边的索引。

```
class MaxFlow {
public:
    ...
    // 使用缩放因子 的DFS寻找增广路径
    bool dfs(int u, int t, int delta, vector<bool>& visited, vector<int>& path) {
        visited[u] = true;
        if (u == t) return true;

        for (int i = 0; i < adj[u].size(); ++i) {
            Edge& e = adj[u][i];
            if (!visited[e.to] && e.capacity - e.flow >= delta) {
                path.push_back(i);
                if (dfs(e.to, t, delta, visited, path)) {
                    return true;
                }
                path.pop_back();
            }
        }
        return false;
    }
};
```

首先确定初始的缩放因子 δ 为所有边容量中的最大值的最大 2 的幂次方。外层循环在 $\delta > 0$ 时执行，内层循环使用 DFS 寻找残余容量不小于 δ 的增广路径。找到增广路径后，计算路径上的最小残余

容量 Δ，并更新路径上所有边的流量和反向边的流量。每次内层循环结束后，将 δ 缩小为原来的一半。

```
class MaxFlow {
public:
    ...
    // 带缩放的Ford-Fulkerson算法
    int fordFulkersonScaling(int s, int t, int totalSourceCapacity) {
        int maxFlow = 0;
        int delta = 1;

        // 设置delta为不大于总源点容量的最大2的幂
        while ((delta << 1) <= totalSourceCapacity) {
            delta <<= 1;
        }

        // 循环减小delta值
        while (delta > 0) {
            vector<bool> visited(n, false);
            vector<int> path;

            // 寻找所有残留容量为delta的增广路径
            while (dfs(s, t, delta, visited, path)) {
                int minResidual = INT_MAX;

                // 计算路径上的最小残留容量
                int u = s;
                for (int index : path) {
                    minResidual = min(minResidual, adj[u][index].capacity -
                        adj[u][index].flow);
                    u = adj[u][index].to;
                }

                // 更新流量
                u = s;
                for (int index : path) {
                    Edge& e = adj[u][index];
                    e.flow += minResidual;
                    adj[e.to][e.reverse].flow -= minResidual;
                    u = e.to;
                }

                maxFlow += minResidual;
                visited.assign(n, false);
                path.clear();
            }

            // 减小delta值
            delta >>= 1;
        }
```

```
47          return maxFlow;
48      }
49  };
```

printGraph() 用于打印图的边信息，包括超级源点到源点、源点到中间节点、中间节点到汇点、汇点到超级汇点的边的容量。这部分代码与算法设计无关，不再贴出。

6 算法测试

6.1 测试样例设计

设计了三种复杂度的测试样例，分别为：

(1) 两个源点，两个汇点，两个中间节点；

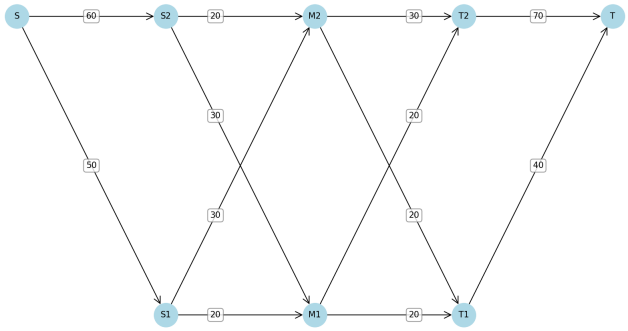

图 6.1: 测试样例 1 示意图

(2) 三个源点，三个汇点，三个中间节点；

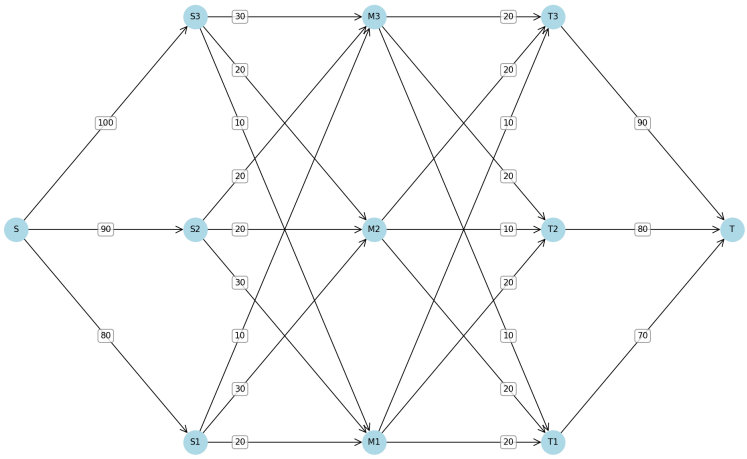

图 6.2: 测试样例 2 示意图

(3) 四个源点，四个汇点，四个中间节点。

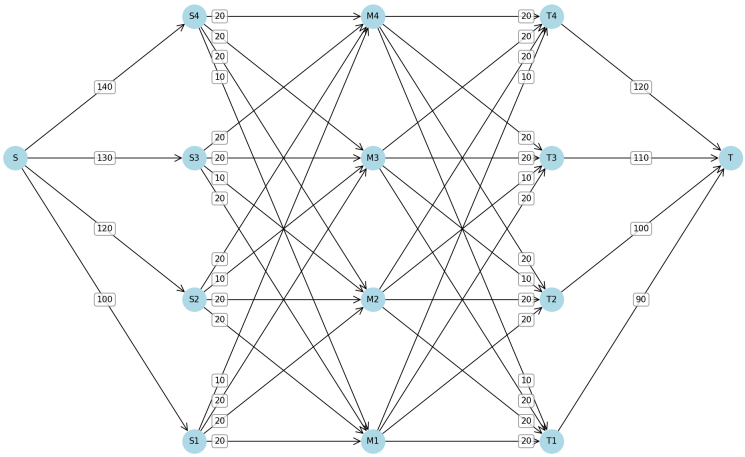

图 6.3: 测试样例 3 示意图

6.2 测试样例初始化

以最简单的测试样例为例，初始化了两个源点、两个汇点、两个中间节点的图。

```
int main() {
    // 测试用例 1
    cout << "测试用例 1:" << endl;
    int numSources1 = 2;
    int numSinks1 = 2;
    int numMedium1 = 2;
    vector<int> sourceCapacities1 = { 50, 60 };
    vector<int> sinkCapacities1 = { 40, 70 };
    vector<vector<int>> sourceToMedium1 = {
        {20, 30},
        {30, 20}
    };
    vector<vector<int>> MediumToSink1 = {
        {20, 20},
        {20, 30}
    };
    ...
}
```

其中，*sourceCapacities*1 表示源点的初始供应，*sinkCapacities*1 表示汇点的最大接收容量，*sourceToMedium*1 表示源点到中间节点的容量，*MediumToSink*1 表示中间节点到汇点的容量。

在这个测试样例里便是，源点 1 的初始供应为 50，源点 2 的初始供应为 60，源点 1 到中间节点 1 的容量为 20，源点 1 到中间节点 2 的容量为 30，源点 2 到中间节点 1 的容量为 30，源点 2 到中间节点 2 的容量为 20，中间节点 1 到汇点 1 的容量为 20，中间节点 1 到汇点 2 的容量为 20，中间节点 2 到汇点 1 的容量为 20，中间节点 2 到汇点 2 的容量为 30，汇点 1 最大接收容量为 40，汇点 2 最大接收容量为 70。当“源点和中间节点”或者“中间节点和汇点”间的容量为 0 时，表示不存在这条有向边。

6.3 测试函数实现

$testCase()$ 构建测试用的图,包括超级源点、源点、中间节点、汇点和超级汇点, 同时会调用 $MaxFlow$ 类的方法添加边、打印图信息、计算最大流和输出初始流量分配方案。这部分代码与算法设计无关，不再贴出。

7 实验结果

```
测试用例 1:
网络流图信息:
超级源点 0 -> 源点 1 容量: 50
超级源点 0 -> 源点 2 容量: 60
源点 1 -> 中间节点 1 容量: 20
源点 1 -> 中间节点 2 容量: 30
源点 2 -> 中间节点 1 容量: 30
源点 2 -> 中间节点 2 容量: 20
中间节点 1 -> 汇点 1 容量: 20
中间节点 1 -> 汇点 2 容量: 20
中间节点 2 -> 汇点 1 容量: 20
中间节点 2 -> 汇点 2 容量: 30
汇点 1 -> 超级汇点 7 容量: 40
汇点 2 -> 超级汇点 7 容量: 70
```

```
最大初始流量: 90 单位

初始流量分配方案:
从源点 1 到中间节点 1 的初始流量: 20 单位
从源点 1 到中间节点 2 的初始流量: 30 单位
从源点 2 到中间节点 1 的初始流量: 20 单位
从源点 2 到中间节点 2 的初始流量: 20 单位
从中间节点 1 到汇点 1 的初始流量: 20 单位
从中间节点 1 到汇点 2 的初始流量: 20 单位
从中间节点 2 到汇点 1 的初始流量: 20 单位
从中间节点 2 到汇点 2 的初始流量: 30 单位
```

图 7.4: 测试样例 1 结果

```
测试用例 2:
网络流图信息:
超级源点 0 -> 源点 1 容量: 80
超级源点 0 -> 源点 2 容量: 90
超级源点 0 -> 源点 3 容量: 100
源点 1 -> 中间节点 1 容量: 20
源点 1 -> 中间节点 2 容量: 30
源点 1 -> 中间节点 3 容量: 10
源点 2 -> 中间节点 1 容量: 30
源点 2 -> 中间节点 2 容量: 20
源点 2 -> 中间节点 3 容量: 20
源点 3 -> 中间节点 1 容量: 10
源点 3 -> 中间节点 2 容量: 20
源点 3 -> 中间节点 3 容量: 30
中间节点 1 -> 汇点 1 容量: 20
中间节点 1 -> 汇点 2 容量: 20
中间节点 1 -> 汇点 3 容量: 10
中间节点 2 -> 汇点 1 容量: 20
中间节点 2 -> 汇点 2 容量: 10
中间节点 2 -> 汇点 3 容量: 20
中间节点 3 -> 汇点 1 容量: 10
中间节点 3 -> 汇点 2 容量: 20
中间节点 3 -> 汇点 3 容量: 20
汇点 1 -> 超级汇点 10 容量: 70
汇点 2 -> 超级汇点 10 容量: 80
汇点 3 -> 超级汇点 10 容量: 90
```

```
最大初始流量: 150 单位

初始流量分配方案:
从源点 1 到中间节点 1 的初始流量: 20 单位
从源点 1 到中间节点 2 的初始流量: 30 单位
从源点 1 到中间节点 3 的初始流量: 10 单位
从源点 2 到中间节点 1 的初始流量: 30 单位
从源点 2 到中间节点 2 的初始流量: 20 单位
从源点 2 到中间节点 3 的初始流量: 20 单位
从源点 3 到中间节点 1 的初始流量: 0 单位
从源点 3 到中间节点 2 的初始流量: 0 单位
从源点 3 到中间节点 3 的初始流量: 20 单位
从中间节点 1 到汇点 1 的初始流量: 20 单位
从中间节点 1 到汇点 2 的初始流量: 20 单位
从中间节点 1 到汇点 3 的初始流量: 10 单位
从中间节点 2 到汇点 1 的初始流量: 20 单位
从中间节点 2 到汇点 2 的初始流量: 10 单位
从中间节点 2 到汇点 3 的初始流量: 20 单位
从中间节点 3 到汇点 1 的初始流量: 10 单位
从中间节点 3 到汇点 2 的初始流量: 20 单位
从中间节点 3 到汇点 3 的初始流量: 20 单位
```

图 7.5: 测试样例 2 结果

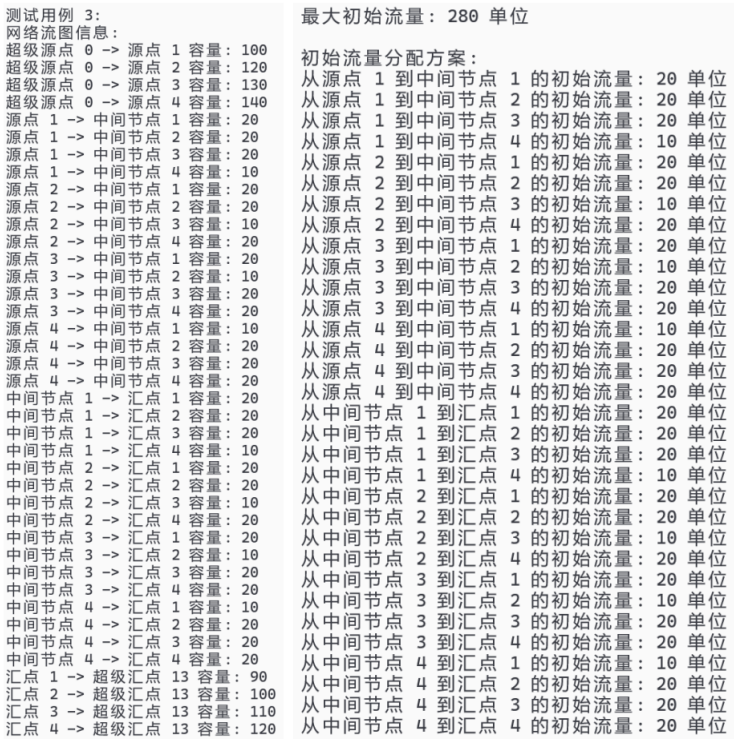

图 7.6: 测试样例 3 结果

画出各测试样例的剩余图，超级源点能够到达的所有点构成集合 A，剩余节点构成 B，此时 $c(A,B)$ 与我们计算所得的结果相等，由最大流最小割定理验证了我们算法的正确性。这说明了各测试用例的流量分配方案能合理分配邮件流量，充分利用各传输通道的容量，有效避免拥堵和资源浪费，实现了邮件输送的最大化目标。

8 总结

本实验研究聚焦多源多汇邮件输送问题，借助图论中的流网络模型，运用带缩放的 $Ford-Fulkerson$ 算法成功求解最大流，为邮件的高效输送提供了优化方案。算法通过 DFS 寻找增广路径，并利用缩放因子 δ 优化搜索过程，有效提升了求解效率，适用于大规模复杂物流网络。实验结果充分验证了算法的正确性和有效性，展现了其在实际物流场景中的应用潜力。然而，研究仍存在可拓展空间。未来可进一步优化算法，以应对动态变化的邮件流量和网络拓扑结构；也可结合其他算法（如 $Dinic$ 算法）进一步提升算法性能；此外，将模型拓展至多目标优化场景（如成本、时间等多因素权衡），可更全面地满足实际物流需求。总之，本研究为邮件输送问题的解决提供了坚实的理论基础和有效的方法工具，对提升物流系统的智能化水平具有重要意义。