山东大学 计算机科学与技术 学院

数据结构与算法课程设计 课程实验报告

学号：202200400053	姓名： 王宇涵	班级： 2202
上机学时：4	实验日期： 2024-05-08	

课程设计题目：

网络放大器

软件开发环境：

Clion 2023.1.1

报告内容：

1. 需求描述

1.1 问题描述

对于一个石油传输网络可由一个加权有向无环图 G 表示。该图中有一个称为源点的顶点 S，从 S 出发，石油流向图中的其他顶点，S 的入度为 0，G 中每条边上的权重为它所连接的两点间的距离。

在输送石油的过程中，需要有一定的压力才能使石油从一个顶点传输到另一个顶点，但随着石油在网络中传输，压力会损失，压力的损失量是传输距离的函数。

为了保证石油在网络的正常运输，在网络传输中必须保证在任何位置的压力都要不小于最小压力 Pmin。为了维持这个最小压力，可在 G 中的一些或全部顶点放置压力放大器，压力放大器可以将压力恢复至该网络允许的最大压力 Pmax。

设 d 为石油从压力 Pmax 降为 Pmin 所走的距离，在无压力放大器时石油可输送的距离不超过 d 的情况下，要求在 G 中放置最少数量的放大器，使得该传输网络从 S 出发，能够将石油输送到图中的所有其他顶点。

为了简化计算，可设每走一个单位长度就会降低一个单位压力且 Pmin=0，即 d=Pmax-Pmin=Pmax。起点处的压力为 Pmax。为了尽可能保证石油的运输（考虑到现实中某些管道可能损坏），首先我们保证所有通道都可以运输，即最大边权值不超过 d；且如果顶点 x 有多条入边，x 处的压力为到达 x 处的压力中的最小值。

1.2 基本要求

1. 设计并实现加权有向无环图的 ADT。
2. 给出两种方法以解决上述问题，验证两种方法的正确性。
3. 比较两种方法的时间和空间性能，用图表显示比较结果。

1.3 输入说明

输入界面设计

第一行包含三个整数 [n, m, d]，设源点 S 为 1 号顶点。

- n 表示顶点个数
- m 表示边的个数
- d 表示石油从压力 Pmax 降为压力 Pmin 所走的距离。

接下来 m 行，每行三个正整数[x, y, c]用来标识一条有向边。

- x 表示起点
- y 表示终点
- c 表示边权 输出

输入样例

4 7 21
1 2 4
1 4 14
1 2 5
1 3 17
2 3 10
2 4 12
3 4 21

1.4 输出说明

输出界面设计

一行一个整数, 表示最少的放大器的个数。

输出样例

1

2. 分析与设计

2.1 问题分析

本次实验要求使用两种方法解决问题, 我分别运用了 dfs 回溯法和优先队列分支定界法来解决问题.
我首先实现了图类的 ADT, 并提供了解决方法 1 和 2, 测试功能完善后, 通过测试样例通过文件输入输出来检验正确性. 成功完成基础实验后, 我使用了 graphviz 来作示意图, 使用了 python 中的 matplotlib 来作折线图, 成功完成了图的可视化和对于时间和空间性能的测量.

2.2 主程序设计

graph.h : 图函数定义和两种方法的实现
main.cpp : 确定输入输出, 检测功能
test.cpp : 用于 oj 检测

2.3 设计思路

我首先实现了图类的 ADT, 并提供了解决方法 1 和 2, 测试功能完善后, 通过测试样例通过文件输入输出来检验正确性. 成功完成基础实验后, 我使用了 graphviz 来作示意图, 使用了 python 中的 matplotlib 来作折线图, 成功完成了图的可视化和对于时间和空间性能的测量.

2.4 数据及数据类型定义

图类

```
class Graph{
public:
    int n, m, pMax; // n 个点, m 条边, pMax 最大压力
    int e[N], h[N], ne[N], idx, w[N]; // 邻接表
    int ans ;// 最终答案
    int inDegree[N],topS[N];// 入度, 拓扑排序
    int p[N]; int tmpP[N];// p[i]表示 i 点的压力, tmpP[i]表示 i 点的临时压力
    double runTime;// 运行时间
    void add(int a, int b, int c) ;// 添加边
    Graph(int n, int m, int pMax) ;// 构造函数
    Graph() {}
    void init();
    bool status[N]; // 放置状态
    bool st[N]; // 选中状态
    void topSort() ;// 拓扑排序
```

```
    void solve1(int index);// 回溯法
    void solve2(int index);// 分支定界法
    void dfs1(int l, int cnt);// 回溯法
    void branchBound();// 分支定界法
    void visualize(int index);// 可视化
    void readFile(string path);// 读取文件
    bool compareFile(string path);// 比较文件
    int mxSize = 0;// 最大堆的大小
};
```

分支定界法所用结点类

```
class Node{
public :
    Node * parent;// 父节点
    int press, lever, cnt;// 压力，层，点的个数
    bool status;// 状态
    Node (Node *parent, int press, int lever, int cnt, bool status) : parent(parent), press(press), lever(lever), cnt(cnt), status(status) {}
    Node () {}
};
```

测量时间

```
    LARGE_INTEGER start_time, end_time;     //开始和结束时间
    double dqFreq;                  //计时器频率
    LARGE_INTEGER freq;             //计时器频率
    QueryPerformanceFrequency(&freq);
    dqFreq = (double)freq.QuadPart;
    QueryPerformanceCounter(&start_time); //计时开始
    Func()
    QueryPerformanceCounter(&end_time); //计时 end
    runTime = (end_time.QuadPart - start_time.QuadPart) / dqFreq * 1000;
```

2.5. 算法设计及分析

主要分析两种解决方法, dfs 回溯法和优先队列分支定界法

Dfs 回溯法

思路 ： 先进行拓扑序列排序，确定结点的先后顺序. 然后我们从先往后枚举结点 u.

先通过 u 的前驱节点计算出 u 本身的 p，再通过 u 计算出 u 的后驱节点的 v 的 p，并判断是否满足运输的要求.

如果 u 本身和出边结点 v 都不能满足运输的要求(即 Pv < 0)，则 u 处放置, Pu 设为 Pmax，并递归到下一个结点.

如果 u 本身或出边结点 v 可以满足运输的要求(即 Pv < 0)，则 1. u 处放置, Pu 设为 Pmax，并递归到下一个结点.2. u 处不放置, Pu 不变，并递归到下一个结点.

如果递归到了最后一层，则更新 ans 为 cnt 的最小值.

优化 ： 通过剪枝的方法进行优化. 如果某一次递归的 cnt 已经 > ans 了，则 return; 其中最初通过拓扑排序和是否满足运输要求的判断已经排除了很多情况.

优先队列分支定界法

思路 ：将所有的状态设置成节点 Node，包含信息为: 遍历到的层数 level，已放置节点个数 cnt，父节点

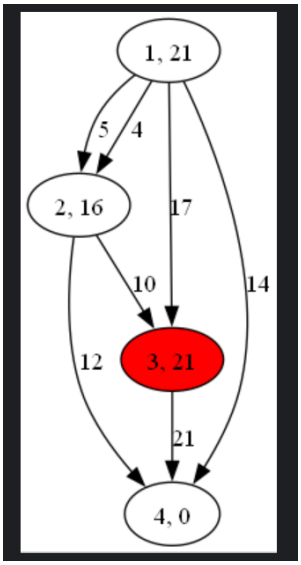
father(用来找前驱结点的 p 值)，放置状态 status，该结点压力 p.
优先队列 heap 按照 cnt 最小值作为队首，并设置一个活动点来保存状态.
进入循环，直到遍历到最后一层为止.
每一次循环，先通过 u 的前驱节点计算出 u 本身的 p，再通过 u 计算出 u 的后驱节点的 v 的 p，并判断是否满足运输的要求.
如果 u 本身和出边结点 v 都不能满足运输的要求(即 Pv < 0)，则将 u 处放置，新建立一个结点 newNode，将 ather 置为活动点, p 置为 pMax, level = 当前 level + 1, cnt = 当前 cnt + 1;
如果 u 本身或出边结点 v 可以满足运输的要求(即 Pv < 0)，则 1. u 处放置，将 father 置为活动点, p 置为 pMax, level = level + 1, cnt = cnt + 1; 2. u 处不放置：将 father 置为活动点, p 不变, level = level + 1, cnt 不变;
每次建立一个新结点，都将新结点 push 进优先队列.
遍历到最后一层结点时，优先队列的队头元素的 cnt 值就是所需答案.
优化：因为我们只需要判断后续结点是否满足要求，则我们不需要每次都遍历后续所有结点，而是加入边的时候就保存一下出边的最大值，这样可以简化时间，以及优先队列的特性，相较于 dfs 不需要遍历所有的情况就可以得出结果.

两种算法的时空复杂度分析
二者的时间复杂度均为 O(2^n)，空间复杂度 dfs 为 O(n)，分支定界为 O(2^n)
由于两者都进行了大量的优化，因此实际的时间大大缩短.

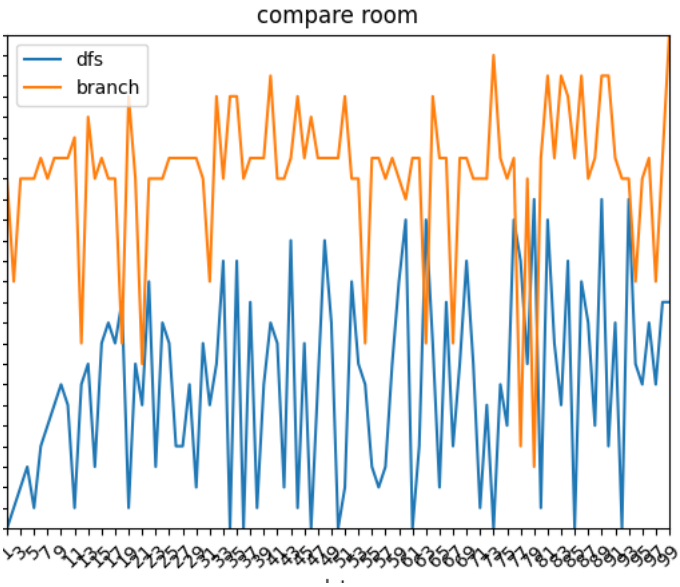
3. 测试

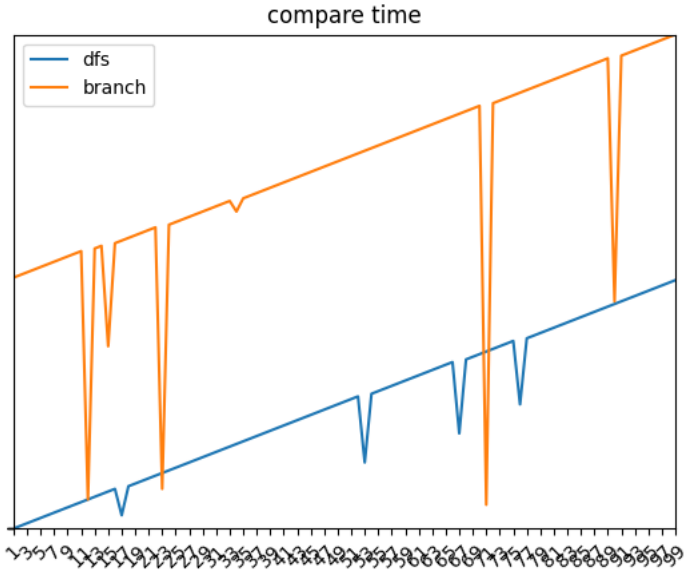
我们首先通过 SDU OJ 进行了测试，发现两种方法均正确

其次，为了实现图的可视化，我学习并实践了 graphviz，最终得到了所有测试样例的图，其中选中放置点由红色标出.



最后，为了比较两者的时空性能，我学习并实践了 python 的 matplotlib，作出了对应的折线图.





发现 branch 的空间占用大于 dfs，而用时相对于 dfs 较多.

前者符合逻辑，因为 branch 的空间复杂度是 O(2^n)，后者与预想有些偏差，猜测原因为 :1. 测试 branch 版本并没有进行出边优化 2. dfs 算法的选择策略相比 branch 在小规模数据中更有优势.

4. 分析与探讨

本次实验我通过 dfs 回溯法先通过了测试样例，此后我采用类似的思路，使用了分支定界法也成功完成了测试，为了分析图的方便和比较二者的性能，我学习并实践了 graphviz 和 python 的 matplotlib，完成了图和性能比较的可视化，成功完成实验.

当然，过程不是一帆风顺的，我也遇到了一些问题.

1. dfs 如何保存最后所有结点的状态信息?

答 : dfs 每次递归的时候且更新 ans 的时候，我都将最终的状态信息更新为当前的状态信息，最后得到的就是最终的 ans 最小时状态信息.

2. 如何实现图的可视化和性能比较的可视化?

答 : 通过查询资料和学习，发现 graphviz 可以简洁地实现图的可视化，而 matplotlib 可以类似 matlab 容易地生成性能比较的折线图.

5. 附录：实现源代码

Dfs 回溯法

```
//回溯法
void Graph::solve1(int index) {
	//测量运行时间
	LARGE_INTEGER start_time, end_time;	//开始和结束时间
	double dqFreq;			//计时器频率
	LARGE_INTEGER freq;		//计时器频率
	QueryPerformanceFrequency(&freq);
	dqFreq = (double)freq.QuadPart;
	QueryPerformanceCounter(&start_time); //计时开始

	topSort();
	status[1] = true;
	p[1] = pMax;
	tmpP[1] = pMax;
	dfs1(2, 0);
	mxSize = n;

	ofstream sfile(to_string(index) + "solve1Size.out", ios::trunc);
	sfile << mxSize << endl;
	QueryPerformanceCounter(&end_time); //计时 end
	runTime = (end_time.QuadPart - start_time.QuadPart) / dqFreq * 1000;
	ofstream ofile(to_string(index) + "solve1y.out", ios::trunc);
	ofile << runTime;
//	visualize(index);
}
//决策目前这个 u 点放置还是不放置
void Graph::dfs1(int l, int cnt) {
	int u = topS[l]; // u 是现在的结点，我们只考虑它对后续结点的影响
	if (l > n) { // 迭代完毕，更新
		ans = min (ans, cnt);
		memcpy(status, st, sizeof st);
```

```
        memcpy(p, tmpP, sizeof tmpP);
        return;
    }

    //找到 u 的前驱节点的最小值
    int pre = INT_MAX;
    for (int k = 1; k <= l - 1; k++) {
        for (int i = h[topS[k]]; i != -1; i = ne[i]) {
            int j = e[i];
            if (j == u) {
                pre = min (pre, tmpP[topS[k]] - w[i]);
            }
        }
    }

    //找到 u 的后驱结点是否存在 < 0 的情况
    bool flag = false;
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (pre - w[i] < 0) {
            flag = true;
            break;
        }
    }

    //两种情况都要放置 u
    if (pre < 0 || flag) {
        st[u] = true;
        tmpP[u] = pMax;
        //剪枝
        if (++cnt >= ans) {
            return;
        }
        dfs1(l + 1, cnt);
    }
    else {
        //可放置可不放置

        //不放置
        st[u] = false;
        tmpP[u] = pre;
        //剪枝
        if (cnt >= ans) {
            return;
        }
        dfs1(l + 1, cnt);
```

```
            //放置
            st[u] = true;
            tmpP[u] = pMax;
            //剪枝
            if (++cnt >= ans) {
                return;
            }
            dfs1(l + 1, cnt);
        }
}
```

优先队列分支定界法

```
//分支定界法
void Graph::solve2(int index) {
        LARGE_INTEGER start_time, end_time;     //开始和结束时间
        double dqFreq;                  //计时器频率
        LARGE_INTEGER freq;             //计时器频率
        QueryPerformanceFrequency(&freq);
        dqFreq = (double)freq.QuadPart;
        QueryPerformanceCounter(&start_time); //计时开始

        topSort();
        branchBound();

        ofstream sfile(to_string(index) + "solve2Size.out", ios::trunc);
        sfile << mxSize << endl;
        QueryPerformanceCounter(&end_time); //计时 end
        runTime = (end_time.QuadPart - start_time.QuadPart) / dqFreq * 1000;
        ofstream ofile(to_string(index) + "solve2y.out", ios::trunc);
        ofile << runTime ;
//      visualize(index);
}
void Graph::branchBound() {
        priority_queue<Node*, vector<Node*>, compare> heap;
        int mx = 0;
        Node *activePoint = new Node(nullptr, pMax, 2, 0, false);
        int level = 2;
        while (level <= n - 1) {
            int u = topS[level];
            int pre = INT_MAX;
            for (int k = 1; k <= level - 1; k++) {
                for (int i = h[topS[k]]; i != -1; i = ne[i]) {
                    int j = e[i];
                    if (j == u) {
                        auto t = activePoint;
                        //将 activePoint  移动到对应的结点上
                        for (int v = level - 1; v > k; v--) {
```

```
                t = t ->parent;
            }
            pre = min(pre, t->press - w[i]);
        }
    }
}

    bool flag = false;
    for (int i = h[u]; i != -1; i = ne[i]) {
        int j = e[i];
        if (pre - w[i] < 0) {
            flag = true;
            break;
        }
    }

    Node *newNode;
    if (pre < 0 || flag) {
        //放置
        newNode = new Node(activePoint, pMax, activePoint->lever + 1, activePoint->cnt + 1, true);
        heap.push(newNode);
    }
    else {
        //不放置
        newNode = new Node(activePoint, pre, activePoint->lever + 1, activePoint->cnt, false);
        heap.push(newNode);
        //放置
        newNode = new Node(activePoint, pMax, activePoint->lever + 1, activePoint->cnt + 1, true);
        heap.push(newNode);
    }

    activePoint = heap.top();
    heap.pop();
    level = activePoint->lever;
    if (heap.size() > mx) {
        mx = heap.size();
    }
}

mxSize = mx;
ans = activePoint->cnt;
auto t = activePoint;
while (t) {
    status[topS[t->lever - 1]] = t->status;
    t = t->parent;
}
```

```
}
```

Graphiz 代码

```
void Graph::visualize(int index) {
	ofstream out("graph" + to_string(index) + ".dot", ios::trunc);
	out<<"digraph G{"<<endl;
	//标志点
	for (int i = 1; i <= n; i++) {
		out<<i<<"[label=\""<<i<<", "<<p[i]<<"\"]"<<endl;
	}
	//连接边
	for (int i = 1; i <= n; i++) {
		//将放置的点整个圆全部标红
		if (status[i]) {
			out<<i<<"[style = filled,fillcolor=red]"<<endl;
		}
		for (int j = h[i]; j != -1; j = ne[j]) {
			int k = e[j];
			out<<i<<"->"<<k<<"[label=\""<<w[j]<<"\"]"<<endl;
		}
	}
	out<<"}"<<endl;
	fclose(stdout);
	string cmd = "dot -Tpng graph" + to_string(index) + ".dot -o graph" + to_string(index) + ".png";
	system(cmd.c_str());
}
```

Matplotlib 代码(以时间比较为例，空间比较与之类似)

```
import matplotlib.pyplot as plt
import numpy
inputText1 = "solve1y.out"
inputText2 = "solve2y.out"
x = []
y1 = []
y2 = []

for i in range(1, 100) :
	x.append(i)
	inputText = str(i) + inputText1;
	with (open(inputText, "r")) as file:
		value = file.readline().strip()
		y1.append(value)
print(x)
print(y1)
for i in range(1, 100) :
	inputText = str(i) + inputText2;
	with (open(inputText, "r")) as file:
		value = file.readline().strip()
```

```
            y2.append(value)
print(y2)

# 绘制折线图
plt.plot(x, y1, label='dfs')
plt.plot(x, y2, label='branch')

# 添加标题和标签
plt.title('Line Plot of Y and Z')
plt.xlabel('X')
plt.ylabel('Y / Z')

y = numpy.arange(0.0, 0.008, 0.001)

plt.xticks(x[0:len(x):2], x[0:len(x):2], rotation=45)
plt.yticks(y, ())
plt.margins(0)
plt.xlabel("data")
plt.ylabel("cost(ms)")
plt.title("compare time")
plt.tick_params(axis="both")
plt.legend()
plt.savefig("TimeCompare.png")
```