算法基础 实验四

PB20151734 陈启思

算法基础 实验四
- 一、实验要求与目的
- 二、实验原理
  - Bellman-Ford算法
  - Dijkstra算法
  - Johnson算法
- 三、实验设备和环境
- 四、方法与步骤
  - 1.生成数据
  - 2.编写图结构和图创建函数
  - 3.实现最小堆优先队列
  - 4.编写Bellman-Ford算法和Dijkstra算法
  - 5.添加代码消除负环
  - 6.编写Johnson算法
  - 7.编写主函数，添加输入、输出的文件操作
  - 8.时间数据可视化
- 五、结果与分析
  - 1.消除负环
  - 2.Johnson算法
    - 1.算法正确性
    - 2.时间分析
- 六、遇到的问题
  - 1.消除负环的算法
  - 2.伪代码描述和具体实现的落差

一、实验要求与目的

本次实验要完成求所有点对的Johnson算法：

题目：

实现Johnson算法，有向图的顶点数分别为27，81，243，729，每个顶点作为起点引出的边的条数取值分别 为：$log_5 N$、$log_7 N$（取下整）。图的输入规模总共有4*2=8个，若同一个N，边的两种规模取值相等，则按后面输出要求输出两次，并在报告里说明。不允许多重边，可以有环。

要求：

本次实验细节较多，关键有如下几点：

- 每种输入规模对应一个输入文件inputxx.txt，**第一行数字为顶点个数，第二行数字为从每个结点出发的边的个数(即出度)**。随后每行有三个数字，分别为**顶点对u、v的序号和边e(u,v)的权重**w_{ij}。
- 顶点序号从1到顶点个数逐个编号，边的权值为$[-10, 50]$间的整数。边的

- 边的终点和权重随机生成。
- 实验开始时，要**首先消除图中的负环**，即对每个负环，删掉其一条边。
- 每种输入规模对应一个输出文件resultxx.txt，**每行列出一个结点对间的最短路径及长度**，例如(1,5,2 20)；若不连通，也要添加说明。
- 撰写实验报告。
- 文件组织需要按照实验要求中的格式，如下图所示：

```
└5-陈启思-PB20151734-project4
    │  report.pdf
    │
    └ex1
        ├input
        │      input11.txt
        │      input12.txt
        │      input21.txt
        │      input22.txt
        │      input31.txt
        │      input32.txt
        │      input41.txt
        │      input42.txt
        │
        ├output
        │      result11.txt
        │      result12.txt
        │      result21.txt
        │      result22.txt
        │      result31.txt
        │      result32.txt
        │      result41.txt
        │      result42.txt
        │      time.txt
        │
        └src
              graph.c
              graph.h
              main.c
```

二、实验原理

Johnson算法用于求所有点对间的最短路径，除了Johnson算法本身的思想，还集成了用于求单源最短路径的Bellman-Ford算法和Dijkstra算法，下面分别简要介绍：

Bellman-Ford算法

对图中**所有边进行$|V|-1$轮松弛**，然后对每条边e(u,v)**检查是否有$v.d > u.d + w(u,v)$成立**，若存在这样的边，说明存在负环，算法返回FALSE；否则返回TRUE，此时每个顶点v的v.d满足$v.d = \delta(s,v)$。伪代码如下：

```
BELLMAN-FORD(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    for i=1 to |G.V|-1
        for each edge(u,v) in G.E
            RELAX(u,v,w)
    for each edge(u,v) in G.E
        if v.d>u.d+w(u,v)
            return FALSE
    return TRUE
```

算法正确性由教材定理24.4保证。

除了以上教材中提到过的内容，**还有一个事实在我的代码中用到**：**若出现$v.d > u.d + w(u,v)$时，说明路径$s \rightsquigarrow u \rightarrow v$上一定有负环**(否则该路径不受负环影响，则一定仍为v.d的上界，矛盾)。这将在消除负环中用到。

Dijkstra算法

Q采用最小堆或二项堆的结构(我用的最小堆)，每次从剩余的集合Q中**提取(并删除)最短路径估计d值最小的顶点**，表示该顶点的最短路径已被找到，并**对该顶点的出边进行松弛更新**：

```
DIJKSTRA(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,s)
    Q=G.V
    while Q is not empty
        u=EXTRACT-MIN(Q)
        for each vertex v in G.Adj[u]
            RELAX(u,v,w)
```

算法正确性由教材定理24.6保证。

Johnson算法

Johnson算法使用**重新赋予权重**技术：若图G中所有边的权重均非负，则对每个结点运行Dijkstra算法即得到所有结点对间最短路径；若包含负边，则采用某种映射，将权重重新赋值，并且保持性质：

- 新权重$\hat{w}$不改变原来的最短路径。
- 所有边的新权重非负。

这样就能将问题等价地转化为第一种情形，即可用Dijkstra算法求解。

于是关键在于权重的映射。由教材409~410页知，只需在原图G中**添加源点s**，使得s到每个结点都有一条权重为0的边；再**令新权重为**$\hat{w}(u,v) = w(u,v) + h(u) - h(v)$，其中$h(u) = \delta(s,u)$表示从源点s到u的最短路径长度，即可满足以上性质。

于是有Johnson算法的伪代码：

```
JOHNSON(G,w)
    compute G',where G'.V=G.V union {s},
        G'.E=G.E union {(s,v):v in G.V},and w(s,v)=0 for all v in G.V
    if BELLMAN-FORD(G',w,s)==FALSE
        print("Negative Cycle!")
    else for each vertex v in G'.V
        set h(v) to the value of δ(s,v)
            computed by the Bellman-Ford algorithm
        for each edge(u,v) in G'.E
            w'(u,v)=w(u,v)+h(u)-h(v)
        let D=(d_{uv}) be a new nxn matrix
        for each vertex u in G.V
            run DIJKSTRA(G,w',u) to compute δ'(u,v) for all v in G.V
            for each vertex v in G.V
                d_uv = δ'(u,v)+h(v)-h(u)
```

```
        return D
```

三、实验设备和环境

编译环境

- 机器主存：16GB
- 处理器主频：1.6GHz
- 语言：C
- IDE：Visual Studio

四、方法与步骤

本次实验代码任务相对前三次实验较多，**具体步骤**可分为：生成数据→编写图结构和图创建函数→实现最小堆优先队列→编写Bellman-Ford算法和Dijkstra算法→添加代码消除负环→编写Johnson算法→编写主函数，添加输入、输出的文件操作→对时间数据可视化并分析。

1.生成数据

由于随机数据的生成不强制使用C/C++，我用python生成了input11.txt等8个输入文件，数据内容和格式均符合题目要求。

2.编写图结构和图创建函数

由于每个顶点的出度均为$\lfloor log_5 N \rfloor$或$\lfloor log_7 N \rfloor$(例如N=27时，出度分别为2,1)，可以视为是**稀疏图**，因此我采用**基于邻接表的图数据结构**，可以有效节省时间、空间。

分别定义顶点ALNode、边ALEdge和图ALGraph的结构体如下：

```c
typedef struct Node{
    int seq;    //结点序号
    int d;      //最短路径估计
    struct Node *pi;     //最短路径中的前驱
    struct Edge *adj;     //邻接表
}ALNode;
typedef struct ALGraph{
    ALNode **vertex;    //顶点的指针数组
    int vcnt;   //顶点数
    int ecnt;   //边数
}ALGraph;
typedef struct Edge{
    struct Edge *next;   //邻接表中的下一条边
    ALNode  *u,*v;     //边的起点和终点
    int weight;
}ALEdge;
```

其中，图ALGraph中主要存储顶点的指针数组，而边可通过顶点中的邻接表域进行访问。

然后是创建图的函数：

```
void create_graph(ALGraph *G, char *filename);
```

传入图G的指针，以及文件名，从文件中读入数据进行顶点和边的创建。其中默认文件数据为题目要求的格式，即先是顶点数、出度，然后每行为起点、终点、权重。

3.实现最小堆优先队列

Dijkstra算法中需要用到基于最小堆的优先队列，因此按照教材第六章的内容、结合图的结构实现该数据结构。堆的定义如下，其中顶点指针作为元素，顶点的属性d为关键字。

```
typedef struct{
    int size;       //堆元素个数
    int length;     //数组元素个数
    int *order;      //order[i]记录seq为i的顶点指针在data中的索引
    ALNode **data;
}heap;
```

需要用到的基本操作有调整、建堆、抽取最小元素、关键字减值，函数API如下：

```
void init_heap(heap *h,ALGraph *G);     //初始化堆
void build_heap(heap *h);       //建立最小堆
void min_heapify(heap *h,int i);    //调整最小堆
void decrease_key(heap* h, ALNode* v, int key);     //减小某节点的关键字
struct Node *extract_min_heap(heap *h); //提取最小元素
```

值得注意的是我在堆中添加了一个**order域**，用于指示序号为i的结点指针在data中的索引，这使得decrease_key()中通过v->seq可以在O(1)时间内得到顶点v在data中的位置。维护order不会增加各个操作的复杂度。

4.编写Bellman-Ford算法和Dijkstra算法

这一步只需根据实验原理中的伪代码进行编写即可。函数API为：

```
int bellman_ford(ALGraph *G, int seq);
void dijkstra(ALGraph *G,int seq);
void _init_single_source(ALGraph *G, int seq);   //求单源最短路径时，进行初始化
```

其中存在负环时，bellman_ford()返回1，否则返回0。

这步中值得注意的是，由于**Dijkstra算法**使用了堆，因此在进行**松弛操作**改变边的属性d时，为了**保证堆的一致性**，要调用**decrease_key()**对属性d进行修改，因此单独为Dijkstra算法定义了一个松弛操作：

```
void relax(ALNode *u, ALNode *v, int w);    //普通松弛操作
void relax_heap(ALNode* u, ALNode* v, int w, heap* h);  //基于最小堆的松弛操作
```

5.添加代码消除负环

虽然只是一个预处理，但我认为这是本实验最为关键、也是相对最难的部分。课上并未讲过相关破负圈的方法，我最终决定**利用在Bellman-Ford算法中提到的事实，在bellman_ford()函数对负环的判断中嵌入破圈的处理**。即**思路为**：若找到顶点u，v，使得$v.d > u.d + w(u, v)$，则从v开始回溯路径$s \leadsto u \to v$，直到找到一个重复出现的顶点，这样就找到了一个负环，只需删除该环中一条边即可。删除边后，仍返回1，表示找到负环；当返回0时，说明不再有负环。

考虑到图可能不连通，模仿Johnson算法，在主函数中先给图增加一个与所有顶点连通的源点s，然后再对顶点s运行Dijkstra算法，消除负边后删除顶点s。这在主函数中完成。

这步仅在bellman_ford()函数和主函数中添加了额外的代码，没有创建新函数来消除负边。

6.编写Johnson算法

主题部分只需要按照实验原理中的伪代码进行编写即可。

但由于实验要求，需要将所有结点对间的最短路径输出到文件中，为了**避免在程序中额外存储$O(|V|^2)$的最短路径数据**(即运行$|V|$次Dijkstra算法、每次得到的$|V|$个结点的前驱pi的数据)，我直接**在johnson()函数中加入了输出文件操作**。每次运行Dijkstra算法后，即时将该源点的最短路径和长度输出到文件中。

由于属性pi用于回溯最短路径，正序输出路径需要使用递归，为此新定义了一个函数_output_path()。函数API如下：

```
void johnson(ALGraph G, int** w, FILE *fp);      //w存储最短路径长度，fp为输出文件的指针
void _output_path(ALNode* t, FILE* fp);   //用于递归输出最小路径
```

按照要求，写入最短路径时，若两点间不连通，也输出一行提示，我采用**形如"(u-/->v) Not Connected"的提示表示不连通。**

7.编写主函数，添加输入、输出的文件操作

主函数中主要**对输入文件迭代**处理，**创建对应的图G**，再添加源点s、一直**调用bellman_ford()消除负环**，直到没有负环；然后**调用函数来运行Johnson算法并计时**，再将时间存入输出文件time.txt。

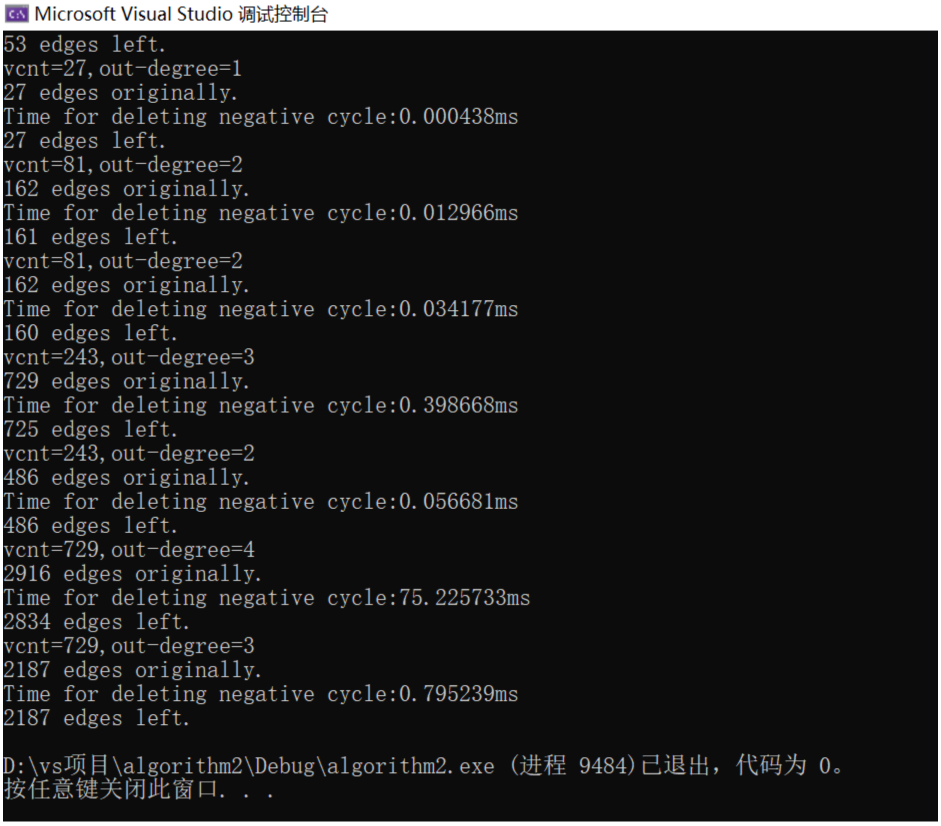
8.时间数据可视化

最后只需对各个数据规模得到的时间进行可视化。该步同样不限语言，我使用python绘制曲线，结果在结果与分析中。

五、结果与分析

1.消除负环

先在终端查看负环消去情况：



```
Microsoft Visual Studio 调试控制台
53 edges left.
vcnt=27,out-degree=1
27 edges originally.
Time for deleting negative cycle:0.000438ms
27 edges left.
vcnt=81,out-degree=2
162 edges originally.
Time for deleting negative cycle:0.012966ms
161 edges left.
vcnt=81,out-degree=2
162 edges originally.
Time for deleting negative cycle:0.034177ms
160 edges left.
vcnt=243,out-degree=3
729 edges originally.
Time for deleting negative cycle:0.398668ms
725 edges left.
vcnt=243,out-degree=2
486 edges originally.
Time for deleting negative cycle:0.056681ms
486 edges left.
vcnt=729,out-degree=4
2916 edges originally.
Time for deleting negative cycle:75.225733ms
2834 edges left.
vcnt=729,out-degree=3
2187 edges originally.
Time for deleting negative cycle:0.795239ms
2187 edges left.

D:\vs项目\algorithm2\Debug\algorithm2.exe (进程 9484)已退出，代码为 0。
按任意键关闭此窗口. . .
```

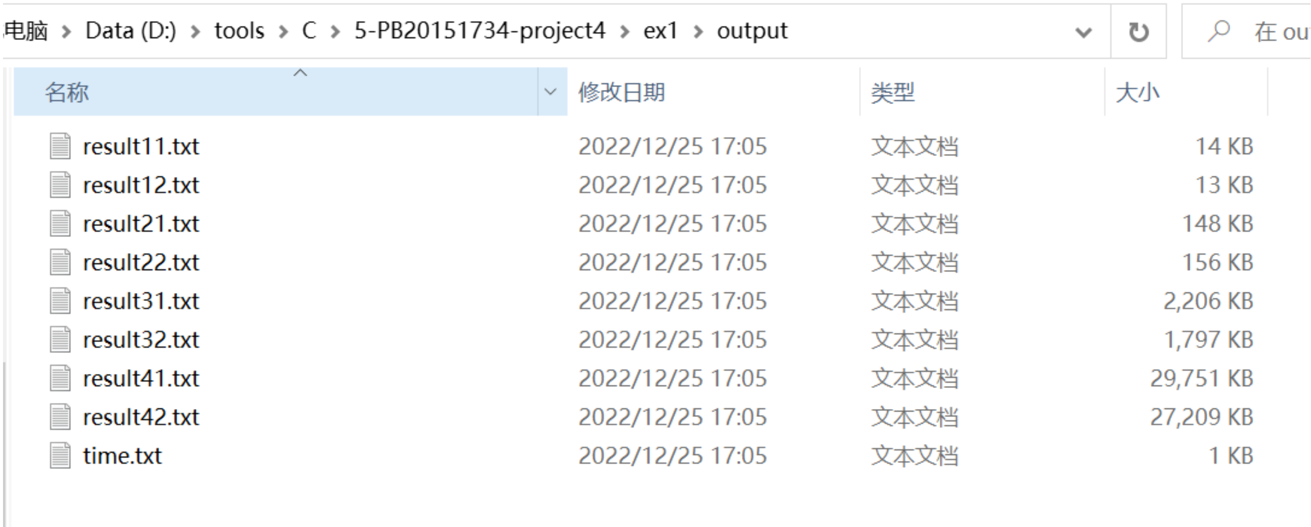
图中输出了对每个输入规模处理前后的边的数目，以及处理用时。例如27个顶点、出度为2时，处理前共54条边，处理后变为53条边，意为原图中恰存在一个负环，且算法有限终止。

规模(顶点数，出度)	27，2	27,1	81,2	81,2	243,3	243,2	729,4	729,3
消去前的边数	54	27	162	162	729	486	2916	2187
消去后的边数	53	27	161	160	725	486	2834	2187

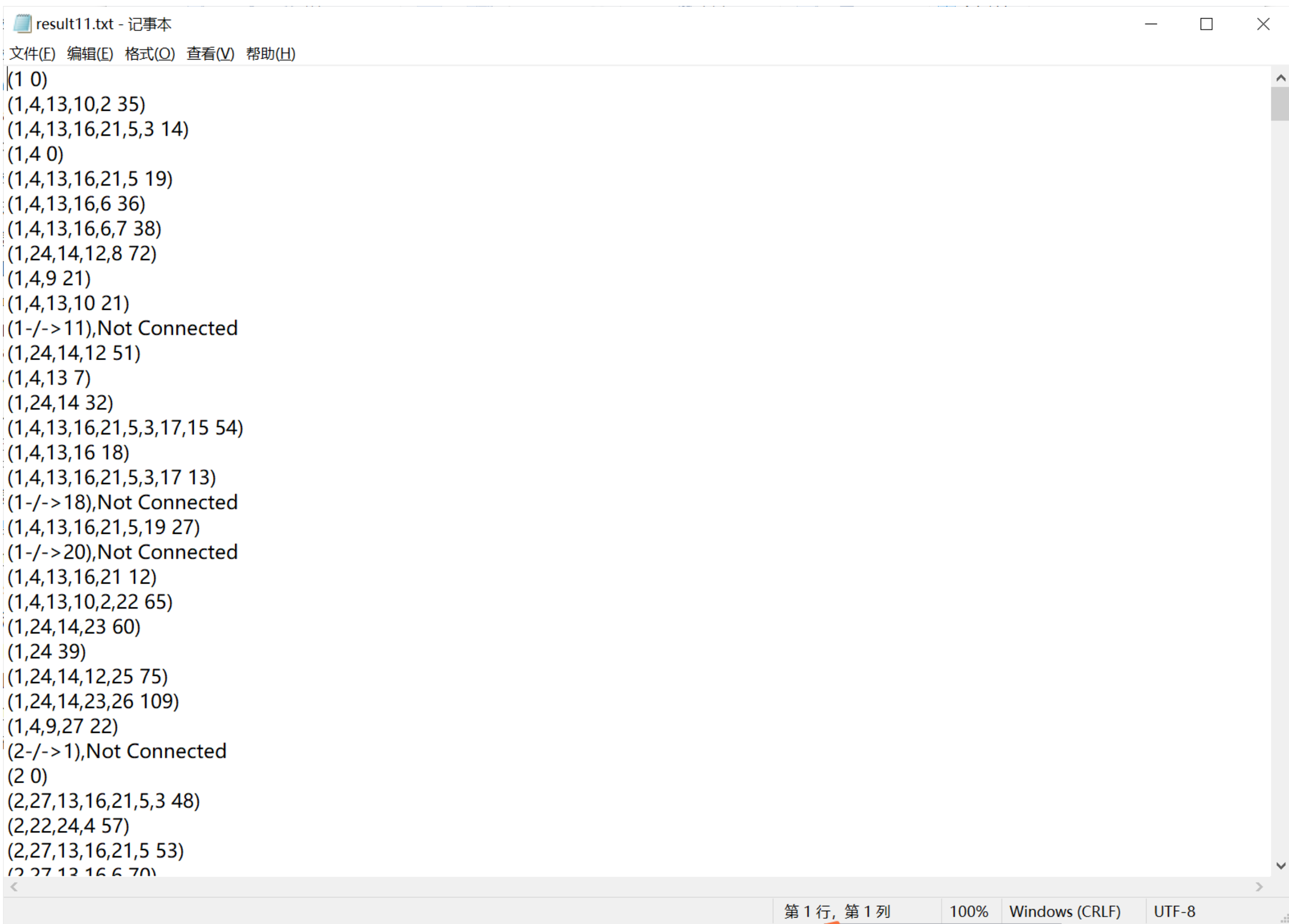
2.Johnson算法

1.算法正确性

如图，按要求生成了8个输出文件和时间数据文件time.txt。



输出文件中，如result11.txt，内容如下：


result11.txt - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
(1 0)
(1,4,13,10,2 35)
(1,4,13,16,21,5,3 14)
(1,4 0)
(1,4,13,16,21,5 19)
(1,4,13,16,6 36)
(1,4,13,16,6,7 38)
(1,24,14,12,8 72)
(1,4,9 21)
(1,4,13,10 21)
(1-/->11),Not Connected
(1,24,14,12 51)
(1,4,13 7)
(1,24,14 32)
(1,4,13,16,21,5,3,17,15 54)
(1,4,13,16 18)
(1,4,13,16,21,5,3,17 13)
(1-/->18),Not Connected
(1,4,13,16,21,5,19 27)
(1-/->20),Not Connected
(1,4,13,16,21 12)
(1,4,13,10,2,22 65)
(1,24,14,23 60)
(1,24 39)
(1,24,14,12,25 75)
(1,24,14,23,26 109)
(1,4,9,27 22)
(2-/->1),Not Connected
(2 0)
(2,27,13,16,21,5,3 48)
(2,22,24,4 57)
(2,27,13,16,21,5 53)
第 1 行，第 1 列 100% Windows (CRLF) UTF-8

其中例如(1,4 0)表示顶点1到顶点4间最短路径为1→4，长度为0；又如(1,4,13 7)表示顶点1到13间最短路径为1→4→13，长度为7；而若两顶点间不连通，例如点1和11，则表示为"(1-/->11),不连通"。

2.时间分析

由教材P411知，**基于二叉最小堆**的Johnson算法的**时间复杂度为**$O(VElgV)$。

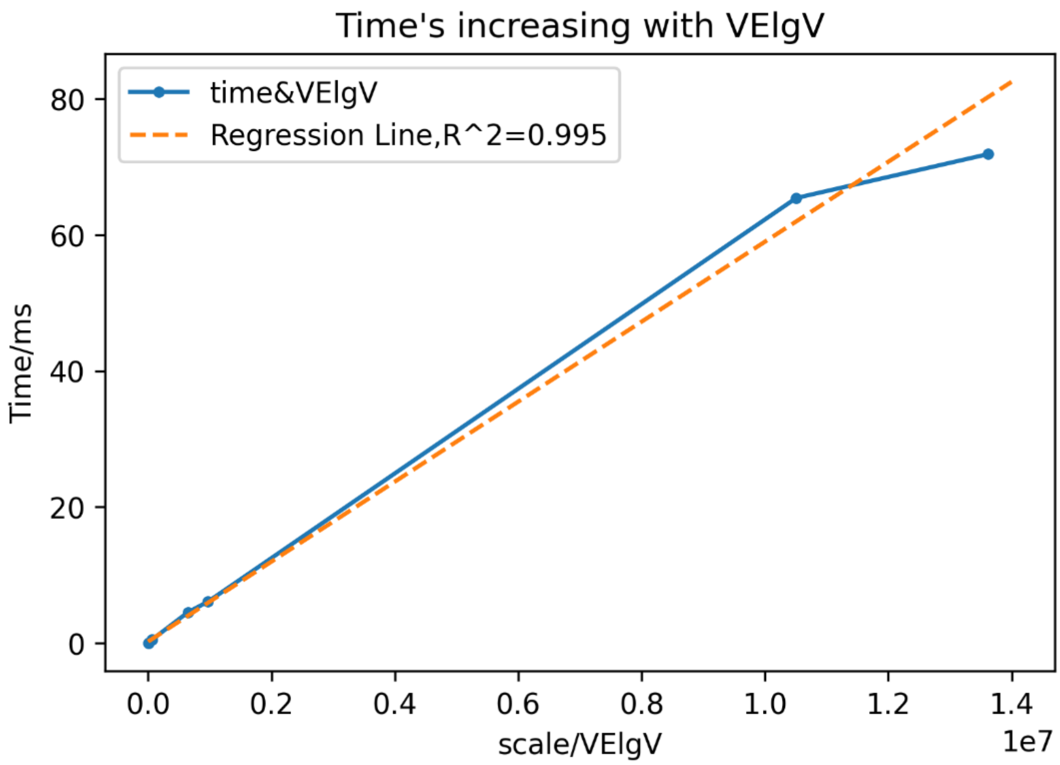
在此列出对8个规模的**消去负环后的**输入(V,E)及其对应的时间，以及**中间统计量VElgV**（用于对时间time进行线性拟合）：

规模(V，E)	27，53	27,27	81,161	81,160	243,725	243,486	729,2834	729,2187
时间/ms	0.036043	0.020153	0.472712	0.518449	6.083808	4.557973	71.843725	65.402628
VElgV	4.716e+03	2.402e+03	5.730e+04	5.695e+04	9.677e+05	6.487e+05	1.361e+07	1.050e+07

注：由于之后我又运行过程序，可能文件中的时间与上述数据有不尽相同，但并不影响分析。

期望：如果算法复杂度满足$O(VElgV)$，那么以$VElgV$为自变量，y为因变量进行线性拟合，应该能得到一条**近似直线**(当$T=\Theta(VElgV)$)或**次线性曲线**(当$T=o(VElgV)$，即相同的$VElgV$值对应着更小的T值)。

根据上表画出Time-VElgV的图像如下，其中蓝色实线是实际数据，**虚线是**对时间Time和横坐标VElgV的**回归直线**。显然，**实际的时间与规模的关系近似于一条直线，且与回归线非常贴近。更进一步，可以算出时间Time与规模VElgV的线性相关系数$R^2\approx 0.995$，基于这些观察和数据，可以断定：我所实现的Johnson算法实际复杂度为O(VElgV)，与理论复杂度一致。**



六、遇到的问题

1.消除负环的算法

一开始我并没有找到正确的算法，我使用了深度优先搜索，通过是否正在访问一个指向灰色顶点的边来判断是否遇到环，再回溯累加权重来计算环的权重，从而判断是否为负边。

但这种做法虽然在顶点数较小时偶然对了，但在input41.txt的数据中发现仍然有负环。我再三思索发现，我并没有找到所有环，因为DFS不会访问黑色结点，但是要找环，通过访问黑色结点也可能存在一个新的环。于是要沿着这种思路修正算法，可能会达到指数级的复杂度，我放弃了这种方法，转而用上述报告中提到的做法。

2.伪代码描述和具体实现的落差

在实现伪代码时，**有些步骤并不是"所见即所得"**，可以直接转换为代码：

- 例如实现Dijkstra算法时，如果沿用原来的松弛操作，直接修改顶点v->d后会导致最小堆的性质被破坏，所以还得修改松弛操作，在要修改v->d时调用decrease_key()来减小v->d才行；
- 还有松弛函数中，判断$v.d\ >\ u.d+w$时，由于我将u.d初始化为INT_MAX，所以当w>0时，$u.d+w$会溢出变成负数，导致判断错误，因此需要先判断$u.d == INT_MAX$是否成立。

以上的错误都是在debug时才发现，这启示我在将伪代码转换为代码时，还要思考具体的数据结构实现，各种操作会产生什么问题，而不是不加思考地直接转译。