lab4图算法

实验内容及要求

- 实验4.1：Kruskal算法
  - 实现求最小生成树的Kruskal算法。无向图的顶点数N的取值分别为：8、64、128、512，对每一顶点随机生成1~[N/2]条边，随机生成边的权重，统计算法所需运行时间，画出时间曲线，分析程序性能。
- 实验4.2：Johnson算法
  - 实现求所有点对最短路径的Johnson算法。有向图的顶点数 N 的取值分别为： 27、81、243、729，每个顶点作为起点引出的边的条数取值分别为：log5N、log7N（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。)　　N=81时会出现这种情况

统计算法所需运行时间，画出时间曲线，分析程序性能。

实验设备及环境

- 编译运行环境
  - Windows10-mingw-w64
  - vscode
  - clion
- 电脑配置

```
计算机名: DESKTOP-NAIGI2Q
操作系统: Windows 10 家庭中文版 64 位 (10.0，版本 18363)
语言: 中文(简体) (区域设置: 中文(简体))
系统制造商: Dell Inc.
系统型号: G7 7588
BIOS: 1.9.0
处理器: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz (12 CPUs), ~2.2GHz
内存: 16384MB RAM
页面文件: 9058MB 已用，10319MB 可用
```

实验方法和步骤

计时函数

- 和lab3相同，采用了`<windows.h>`中的`QueryPerformance`来计时，精度高，避免了之前数据量较小时时间为0的情况
- 开始计时

```
LARGE_INTEGER t1, t2, tc;
QueryPerformanceFrequency(&tc);
QueryPerformanceCounter(&t1);
```

- 运行需要计时的部分
- 结束计时并计算用时

```
QueryPerformanceCounter(&t2);
double time = (double)(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart;
```

Kruskal算法

主要用于**无向图**

顶点编号从0开始

generate.cpp

- 功能：生成符合要求的输入文件
- 要求：
  - 无向图的顶点数N的取值分别为： 8、64、128、512
  - 对每一顶点随机生成1~[N/2]条边，随机生成边的权重
  - 每行存放一对结点i,j序号（数字表示）和wij，表示结点i和j之间存在一条权值为wij边，权值范围为[1,20]，取整数
  - 如果后续结点的边数大于[N/2]，则无需对该结点生成边
  - 不允许重边，可以有环（包括自环，算作一条边，对应1度）
- 因为这里的N都是偶数，故不用专门进行下取整处理
- 用动态的`v_degree`数组保存当前状态下顶点的出度，用来控制每个顶点的度数小于等于[N/2]
- 用`vector<pair<int, pair<int, int>>> edges`保存生成的所有边，对应`pair<w, pair<u, v>>`，即记录边的权重以及首尾结点
- 循环遍历每个顶点，在每一轮循环中：
  - 根据当前顶点的度数确定对其新增的边数
    - 如果度数小于[N/2]，则对应生成的边数为`1 + rand() % (N / 2 - v_degree[i])`
    - 如果度数等于[N/2]，则对应生成的边数为0
  - 用`vector<pair<int, int>> edges_without_weight`来记录已经生成的边，防止生成重边（因为重边可能权重不同，故这里新建了一个不包含权重的边的容器）
    - 因为是无向图，故一轮生成后将`edge_temp1`和`edge_temp2`

      ```
      edge_temp1.first = i;
      edge_temp1.second = rand() % N;
      edge_temp2.first = edge_temp1.second;
      edge_temp2.second = edge_temp1.first;
      ```

      两个边放入`edges_without_weight`容器中，这样才能完全避免重边
    - 只有不是重边 且 当前顶点度数小于N/2时才生成新的边

      ```
      if(count(edges_without_weight.begin(), edges_without_weight.end(),
      edge_temp1) == 0 &&
      v_degree[edge_temp1.second] < N / 2)
      ```

      否则计数变量不变，`continue`进入下一轮循环

MST-KRUSKAL.cpp

- 分离集合数据结构类（使用21.3节所讨论的不相交集合森林实现，并增加按秩合并和路径压缩的功能）

```
class Disjoint_Set{
public:
    int *p;
    int *rank;
    int n;
    //构造函数，对每个结点x进行Make-Set操作
    Disjoint_Set(int n){
        this->n = n;
        p = new int[n];
        rank = new int[n];
        for(int i = 0; i < n; ++i){
            p[i] = i;
            rank[i] = 0;
        }
    }
    int Find_Set(int x){
        if(x != p[x]){
            p[x] = Find_Set(p[x]);
        }
        return p[x];
    }

    //将课本上的Union和Link放在一起
    void Union(int x, int y){
        x = Find_Set(x);
        y = Find_Set(y);
        if(rank[x] > rank[y]){
            p[y] = x;
        }else{
            p[x] = y;
            if(rank[x] == rank[y]){
                rank[y]++;
            }
        }

    }
};
```

- 无向图类
  - 构造函数：记录顶点数
  - 记录图的所有边（包括首尾结点和权重）

    ```
    vector<pair<int, pair<int, int>>> Edge;
    ```

    用这种方式表示图可以更方便地遍历图中的所有边
  - 向图中添加边的方法

```
    void Add_Edge(int u, int v, int w){
        Edge.push_back({w, {u, v}});
    }
```

- 计算最小生成树的`MST_Kruskal`方法

```
    vector<pair<int, pair<int, int>>> MST_Kruskal(){
        vector<pair<int, pair<int, int>>> A;
        int min_sum_weight = 0;
        Disjoint_Set my_Disjoint_set(V_num);
        sort(Edge.begin(), Edge.end());

        for(auto edge : Edge){
            int u = edge.second.first;
            int v = edge.second.second;
            int u_set_rep = my_Disjoint_set.Find_Set(u);
            int v_set_rep = my_Disjoint_set.Find_Set(v);
            if(u_set_rep != v_set_rep){
                A.push_back({edge.first ,{u, v}});
                min_sum_weight += edge.first;
                my_Disjoint_set.Union(u_set_rep, v_set_rep);
            }
        }
        cout << "min_sum_weight: " << min_sum_weight << endl;
        outfile << "min_sum_weight: " << min_sum_weight << endl;
        return A;
    }
```

  - 主题思路和书上的算法相同，本质上是一个贪心算法
  - 该算法找到安全边的方法：在所有连接森林中两个不同的树的边里面，找到权重最小的边(u, v)
  - 主要是通过for循环，按照权重从低到高的次序对每条边逐一进行检查。对于每条边(u, v)，该循环将检查端点u和v是否属于同一颗树。如果是，该边不能加入到森林中（否则会形成环路）；如果不是，则两个端点分别属于不同的树，然后该边被加入集合，然后将两棵树中的结点进行合并
  - 用`C++`的库函数`sort`来对含权重的边进行排序

    ```
    sort(Edge.begin(), Edge.end());
    ```

- `result`函数的功能：
  - 从输入文件中读入图的信息
  - 通过`Graph`类构建图
  - 统计时间
  - 将结果和时间输出到对应文件

Johnson算法

- 主要用于**有向图**
- 顶点编号从1开始，0号顶点预留给扩展后的G'
- **注意**：为了不统计输出到文件和缓冲区所需要的时间，需要把`Dijkstra`和`Johnson`末尾的输出部分注释掉，以产生合理的`time.txt`。

    - 提交版本的`time.txt`文件包含了输出到文件和缓冲区所需要的时间，作图时用到的数据是将`Dijkstra`和`Johnson`末尾的输出部分注释掉后产生的
- 最大权值为`INT16_MAX`，即32367

generate.cpp

- 功能：生成符合要求的输入文件
- 要求：
    - 有向图的顶点数 N 的取值分别为: 27、81、243、729
    - 每个顶点作为起点引出的边的条数取值分别为： log5N、 log7N（取下整）
    - 每行存放一对结点i,j序号（数字表示）和wij，表示存在一条结点i指向结点j的边，边的权值为wij ，权值范围为[0,50],取整数。
    - 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 。N=81时会出现这种情况
    - 不允许重边，可以有环（包括自环，算作一条边，对应1度）
- 由于权重改为了[0,50]，故`Bellman-Ford`算法返回值一定为真，不用考虑负环的消除
- 采用和`Kruskal`类似的生成思路，只不过限制更少
- 每个顶点出度确定，不再需要记录
- 同样循环遍历每个顶点，生成对应数量的边
    - 用`vector<pair<int, int>> edges_without_weight`来记录当前顶点以及有的出边，防止出现多重边

Johnson.cpp

- 采用邻接表来存储图
    - 边结点类

```
class Edge_Node{
public:
    int src;
    int dest;
    int w;
    Edge_Node *next;
};
```

    - 顶点类

```
class Vertex_Node{
public:
    int data;    //顶点信息，这里主要是编号
    int d;     //相当于v.d，即从源结点s到该结点的最短路径权重的上界
    Vertex_Node* pi;     //相当于v.π，记录当前结点的前驱结点
    Edge_Node *first_edge;
};
```

    - 图类
        - 构造函数：初始化顶点数和输入输出路径
        - `Add_Edge`：向邻接表中插入有向边(u,v)，权重为w

```
void Add_Edge(int u, int v, int w){
            //先分配空间，构造一个新的边结点
```

```
            Edge_Node* temp_edge = new Edge_Node;
            temp_edge->src = u;
            temp_edge->dest = v;
            temp_edge->next = NULL;
            temp_edge->w = w;
            //寻找合适的插入位置
            Edge_Node *u_first = vertex_list[u].first_edge;
            //若当前顶点邻接表为空
            if(u_first == NULL){
                vertex_list[u].first_edge = temp_edge;
            }else{
                while(u_first->next != NULL){
                    u_first = u_first->next;
                }
                u_first->next = temp_edge;
            }
            edge_num++;
        }
```

- `Create_test_G`：输出课本中的例题便于测试
- `Create_G`：从文件中读取并构建图

```
        void Create_G(){
            ifstream infile;
            infile.open(inpath);
            cout << "Read G from " << inpath << endl;
            int u, v, w;
            while(infile >> u >> v >> w){
                Add_Edge(u, v, w);
            }
            infile.close();
        }
```

- `Print_Path_u_to_v`：输出两个顶点的最短路径和权重

```
        void Print_Path_u_to_v(int u, int v){

            cout << "From " << vertex_list[u].data << " to " <<
vertex_list[v].data << endl;
            outfile << "From " << vertex_list[u].data << " to " <<
vertex_list[v].data << " : (";
            Print_Path(&vertex_list[u], &vertex_list[v]);
            cout << "Weight: " << vertex_list[v].d << endl;
            cout << endl;
            outfile << " " << vertex_list[v].d;
            outfile << ")" << endl;

        }
```

- `Print_Path`：`Print_Path_u_to_v`的子过程，用以递归打印路径

```
        void Print_Path(Vertex_Node* u, Vertex_Node* v){
            if(u == v) {
                cout << v->data << ",";
                outfile << v->data << ",";
            }else if(v->pi == NULL){
                cout << "disconnected" << endl;
                outfile << "disconnected";
            }else{
                Print_Path(u, v->pi);
                cout << v->data << " ";
                outfile << v->data << ",";
            }
        }
```

- `Initialize_Single_Source`：同课本，对单源路径进行初始化

```
        void Initialize_Single_Source(int s){
            //对所有顶点进行初始化
            for(int i = 0; i < vertex_num; ++i){
                vertex_list[i].d = MY_MAX;
                vertex_list[i].pi = NULL;
            }
            vertex_list[s].d = 0;
        }
```

- `Relax`：同课本，对边进行松弛操作

```
        void Relax(Edge_Node *edge){
            auto u = vertex_list[edge->src];
            auto v = vertex_list[edge->dest];
            int w_u_v = edge->w;
            if(v.d > u.d + w_u_v){
                vertex_list[edge->dest].d = vertex_list[edge->src].d
+ w_u_v;
                vertex_list[edge->dest].pi = &vertex_list[edge-
>src];
            }
        }
```

- `Bellman_Ford`：同课本，通过对边进行松弛操作来渐进地降低从源结点s到每个结点v的最短路径的估计值v.d，直到该估计值与实际的最短路径权重相同为止。该算法返回真当且仅当输入图不包含可以从源结点到达的权重为负值的环路
  - 对图的每条边进行`|V| - 1`次处理，每次处理都是对边进行一次松弛操作
  - 然后检查图中是否存在权重为负值的环路并返回对应的布尔值

```
        bool Bellman_Ford(int s){
            Initialize_Single_Source(s);
            //对图的每条边作|V| - 1次处理
            for(int i = 1; i < vertex_num; ++i){
                //遍历所有边
                for(int j = 0; j < vertex_num; ++j){
                    Edge_Node *edge = vertex_list[j].first_edge;
                    while(edge != NULL){
                        Relax(edge);
```

```
                    edge = edge->next;
                }
            }
        }
        //检查是否有负环路
        for(int i = 0; i < vertex_num; ++i){
            Edge_Node *edge = vertex_list[i].first_edge;
            while(edge != NULL){
                if(vertex_list[edge->dest].d > vertex_list[edge->src].d + edge->w){
                    return false;
                }
                edge = edge->next;
            }
        }
        cout << "Bellman-Ford finished" << endl;
        return true;
    }
```

- `Dijkstra`：同课本，重复从V-S中选择最短路径估计最小的结点u，将u加入到集合S，然后对所有从u出发的边进行松弛
  - 使用最小优先队列来保存结点集合
  - ```
    void Dijkstra(int s){
        //注意必须先初始化再入队列，否则第一次取出来的就不好说了，或者说肯定不是s
        Initialize_Single_Source(s);
        vector<Vertex_Node> S;
        //vector<Vertex_Node*> node_ptr;
        //优先队列实现最小堆
        priority_queue<int, vector<Vertex_Node*>, cmp> node_min_heap;
        for(int i = 0; i < vertex_num; ++i){
            node_min_heap.push(&(vertex_list[i]));
        }
        while(!node_min_heap.empty()){
            Vertex_Node *min_d_node = node_min_heap.top();
            node_min_heap.pop();
            S.push_back(*min_d_node);
            Edge_Node *edge = min_d_node->first_edge;
            while(edge != NULL){
                Relax(edge);
                edge = edge->next;
            }
        }

        cout << "Dijkstra result: " << endl;

        for(int i = 1; i < vertex_num; ++i){
            if(i != s){
                Print_Path_u_to_v(s, i);
            }
        }
    }
    ```
    ```

- `Johnson`：采用重新赋予权重技术，调用`Bellman-Ford`算法和`Dijkstra`算法来计算所有结点对之间的最短路径，并计时
  - 对G进行扩展为G'
  - 在G'上调用`Bellman-Ford`算法
  - 根据`Bellman-Ford`算法的结果设置h(v)
  - 重新计算权重函数$\hat{w}$
  - 对每一对结点u, v，调用`Dijkstra`算法计算最短路径权重，并将其保存在D矩阵对应位置

```
        void Johnson(){

            LARGE_INTEGER t1, t2, tc;
            QueryPerformanceFrequency(&tc);
            QueryPerformanceCounter(&t1);

            outfile.open(outpath);
            Create_G();
            //Create_test_G();
            //扩展G为G'
            for(int i = 1; i < vertex_num; ++i){
                Add_Edge(0, i, 0);
            }

            if(Bellman_Ford(0) == false){
                cout << "the input graph contains a negative-weight
cycle" << endl;
            }else{
                int *h = new int[vertex_num];
                //set h(v) to the value of sigma(s,v) computed by
the Bellman-Ford algorithm
                for(int i = 0; i < vertex_num; ++i){
                    h[i] = vertex_list[i].d;
                }
                for(int i = 0; i < vertex_num; ++i){
                    Edge_Node* edge = vertex_list[i].first_edge;
                    while(edge != NULL){
                        edge->w = edge->w + h[edge->src] - h[edge-
>dest];
                        edge = edge->next;
                    }
                }
                int **D = new int*[vertex_num];
                for(int i = 0; i < vertex_num; ++i){
                    D[i] = new int[vertex_num];
                }
                for(int i = 1; i < vertex_num; ++i){
                    Dijkstra(i);
                    for(int j = 1; j < vertex_num; ++j){
                        D[i][j] = vertex_list[j].d + h[j] - h[i];
                    }
                }

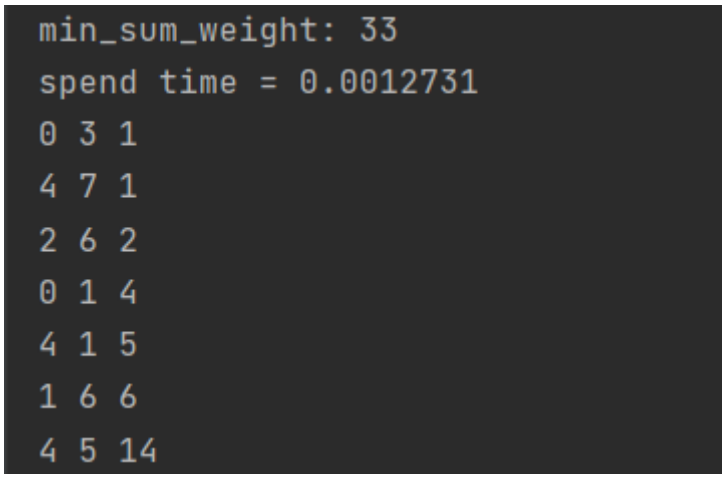
                cout << "Johnson result: " << endl;
                for(int i = 1; i < vertex_num; ++i){
                    for(int j = 1; j < vertex_num; ++j){
                        cout << D[i][j] << " ";
```

```
                    }
                    cout << endl;
                }
            }
            QueryPerformanceCounter(&t2);
            double time = (double)(t2.QuadPart - t1.QuadPart) /
(double)tc.QuadPart;
            cout << "spend time = " << time << endl; //输出时间（单
位： s ）
            outfile_time << time << endl;
            outfile.close();
        }
```

实验结果和分析

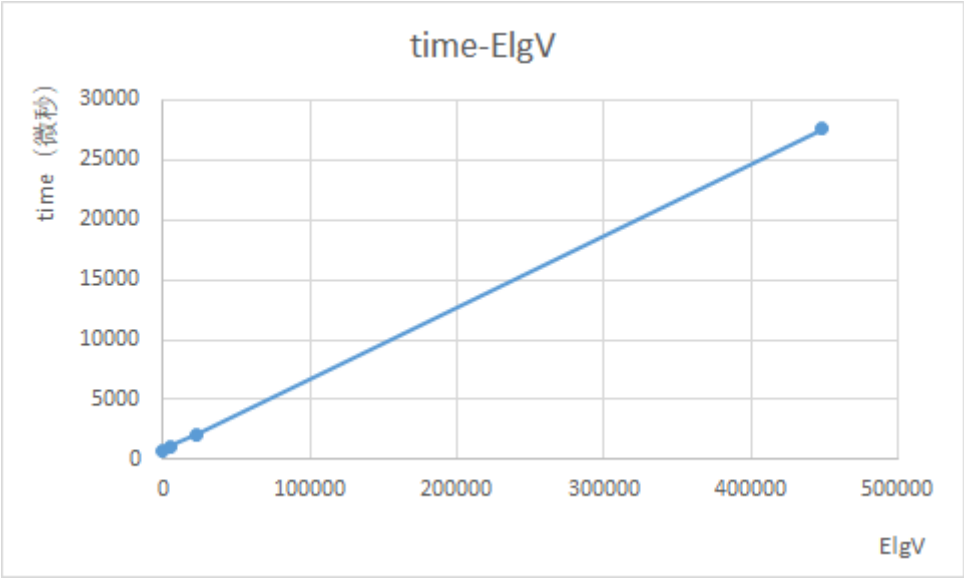
Kruskal算法

- 实验结果截图



- 由于使用了21.3节所讨论的不相交集合森林实现，并增加按秩合并和路径压缩的功能，故按照课本中的推导，理论时间复杂度应为$O(ElgV)$
- 数据和作图如下

V	lgV	E	ElgV	time（微秒）
8	3	18	54	802
64	6	789	4734	1080
128	7	3334	23338	2092
512	9	49828	448452	27588



可以看到实际时间复杂度和理论时间复杂度很符合

Johnson算法

- **N=81时对应的图规模相同**
- 实验结果截图

  这里去除了`Johnson`和`Dijkstra`的输出部分（以免统计时间时不准确，把输入输出的时间也错误地统计进去）

```
D:\study\algorithm_lab\lab4-2\cmake-build-debug\lab4_2.exe
Read G from ../input/input11.txt
Bellman-Ford finished
spend time = 0.0008882
Read G from ../input/input12.txt
Bellman-Ford finished
spend time = 0.0006578
Read G from ../input/input21.txt
Bellman-Ford finished
spend time = 0.0039214
Read G from ../input/input22.txt
Bellman-Ford finished
spend time = 0.0039851
Read G from ../input/input31.txt
Bellman-Ford finished
spend time = 0.0337275
Read G from ../input/input32.txt
Bellman-Ford finished
spend time = 0.0326274
Read G from ../input/input41.txt
Bellman-Ford finished
spend time = 0.33475
Read G from ../input/input42.txt
Bellman-Ford finished
spend time = 0.314208
```

未去除`Johnson`和`Dijkstra`的输出部分

```
Read G from ../input/input11.txt
Bellman-Ford finished
Dijkstra result:
From 1 to 2
disconnected
Weight: 32767

From 1 to 3
1,3 Weight: 1

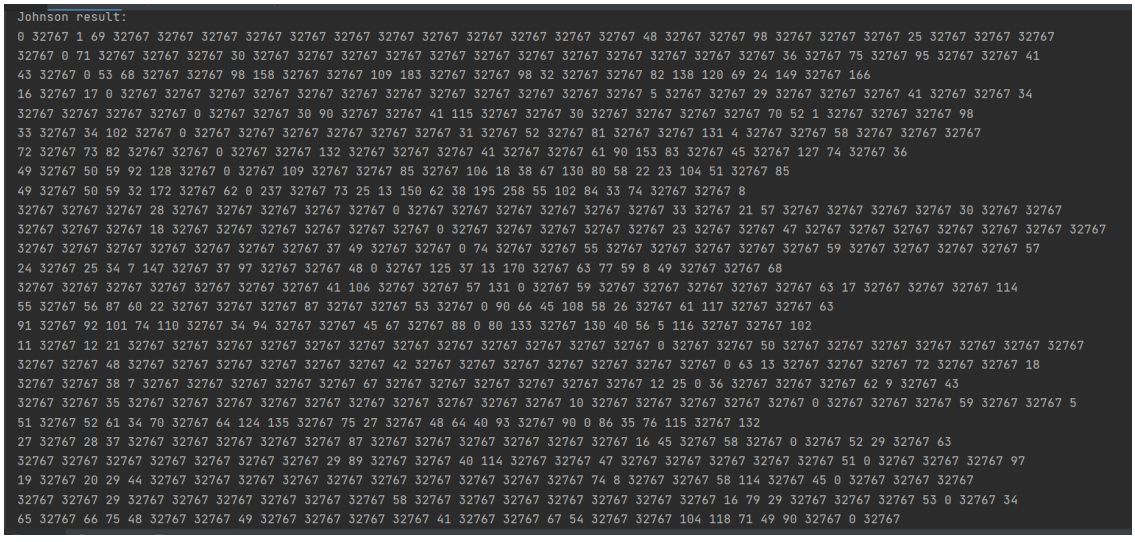
From 1 to 4
1,17 4 Weight: 69

From 1 to 5
disconnected
Weight: 32767

From 1 to 6
disconnected
Weight: 32767

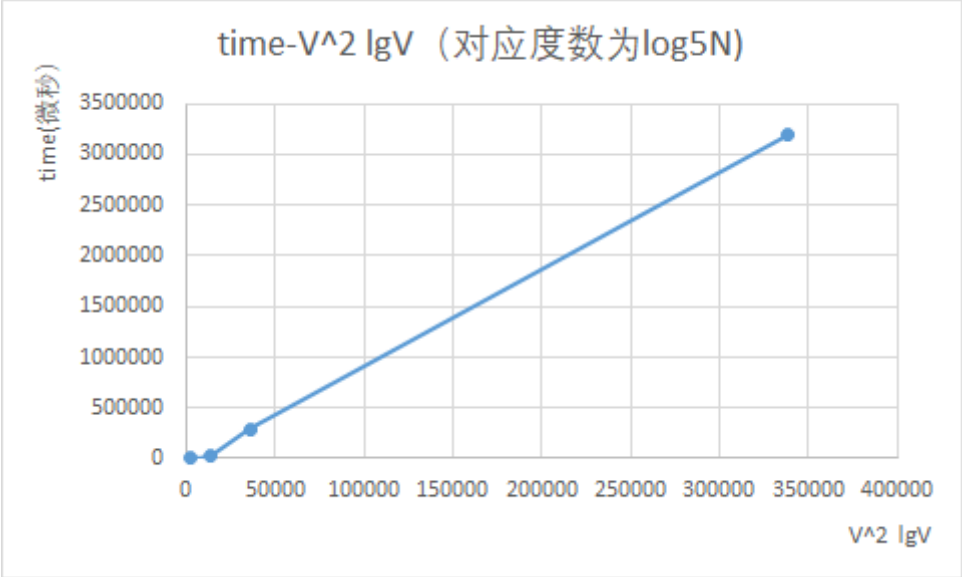
From 1 to 7
disconnected
Weight: 32767
```

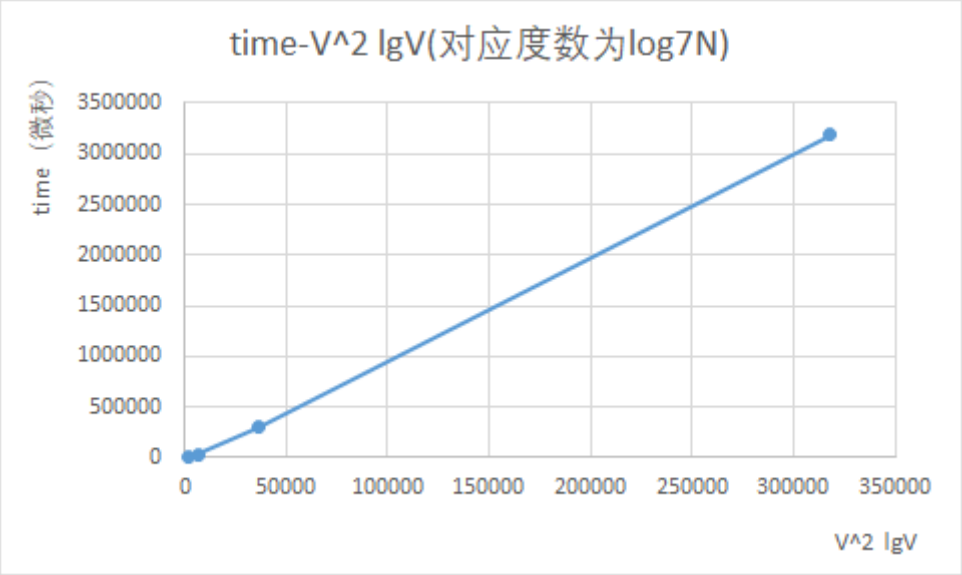
`Johnson` 算法得到的D矩阵



- 由于 `Johnson` 算法调用的 `Dijkstra` 算法部分使用了 `C++` 中的优先队列来实现，由于优先队列本质上是一个堆，不论其是二叉堆还是斐波那契堆，由于实验中 `E=kV`，理论时间复杂度均为 $O(V^2 lgV)$
- 数据和作图如下（对每个顶点引出$log_5 N$和$log_7 N$的两种情况分别作图）

V	lgV	E	V^2 lgV		time（秒）	time（微秒）
27	3	54	2187		0.002884	2883.7
27	3	27	2187		0.001907	1906.8
81	4	162	26244		0.013188	13188.1
81	4	162	26244		0.006615	6614.9
243	5	729	295245		0.035787	35787.2
243	5	486	295245		0.036123	36122.5
729	6	2916	3188646		0.338213	338213
729	6	2187	3188646		0.318075	318075





可以看到实际时间复杂度和理论时间复杂度很符合

实验总结

- 通过本次实验，对最小生成树`Kruskal`算法有了更加深入的了解。同时也手动实现了21章的分离集合数据结构，该数据结构可以使得`Kruskal`算法的时间复杂度较低（目前已知的渐进最快的实现方式）
- 对求所有结点对最短路径的`Johnson`算法有了更深入的了解，同时也实现了求单源最短路径的`Bellman-Ford`算法和`Dijkstra`算法，掌握了这些算法之间的区别和联系
- 让我意识到了图可以有多种存储方式，不同的存储方式对于不同的算法可能更加方便快捷，这是一种空间、时间、算法间的权衡