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。)

■实验4.1 kruskal算法

□ex1/input/

- 每种输入规模分别建立txt文件，文件名称为input1.txt, input2.txt, ……, input4.txt ；
- 生成图的信息分别存放在对应数据规模的txt文件中
- 每行存放一对结点i, j序号（数字表示）和w_{ij}，表示结点i和j之间存在一条权值为w_{ij}边，权值范围为[1, 20]，取整数。
- Input文件中为随机生成边以及权值，每个结点至少有一条边，至多有⌊N/2⌋边，即每条结点边的数目为1 + rand()%⌊$N/2$⌋。如果后续结点的边数大于⌊N/2⌋，则无需对该结点生成边。

□ex1/output/

- result.txt:输出对应规模图中的最小生成树总的代价和边集，不同规模写到不同的txt文件中，因此共有4个txt文件，文件名称为result1.txt, result2.txt, ……, result4.txt;输出的边集要表示清楚，边集的输出格式类似输入格式。

■实验4.2 Johnson算法

□ex2/input/

- 每种输入规模分别建立txt文件，文件名称为input11.txt, input12.txt, ……, input42.txt （第一个数字为顶点数序号（27、81、243、729 ），第二个数字为弧数目序号（ $\log_5 N$、$\log_7 N$ ））；
- 生成的有向图信息分别存放在对应数据规模的txt文件中；
- 每行存放一对结点i, j序号（数字表示）和w_{ij}，表示存在一条结点i指向结点j的边，边的权值为w_{ij}，权值范围为[-10, 50]，取整数。
- Input文件中为随机生成边以及权值，实验首先应判断输入图是否包含一个权重为负值的环路，如果存在，删除负环的一条边，消除负环，实验输出为处理后数据的实验结果，并在实验报告中说明。

□ex2/output/

- result.txt:输出对应规模图中所有点对之间的最短路径包含结点序列及路径长，不同规模写到不同的txt文件中，因此共有8个txt文件，文件名称为result11.txt, result12.txt, ……, result42.txt;每行存一结点的对的最短路径，同一最短路径的结点序列用一对括号括起来输出到对应的txt文件中，并输出路径长度。若图非连通导致节点对不存在最短路径，该节点对也要单独占一行说明。
- time.txt:运行时间效率的数据，不同规模的时间都写到同个文件。
- example:对顶点为27，边为54的所有点对最短路径实验输出应为：(1, 5, 2 20) (1, 5, 9, 3 50)……，执行结果与运行时间的输出路径分别为：
  - output/result11.txt
  - output/time.txt

实验设备和环境

- 编译运行环境

    - Windows10-mingw-w64
    - Clion 2020.2.4
    - vscode
- 硬件
    - 处理器: 英特尔 Core i7-8750H @ 2.20GHz 六核
    - 速度 2.21 GHz (100 MHz x 22.0)
    - 处理器数量 核心数: 6 / 线程数: 12
    - 一级数据缓存 6 x 32 KB, 8-Way, 64 byte lines
    - 一级代码缓存 6 x 32 KB, 8-Way, 64 byte lines
    - 二级缓存 6 x 256 KB, 4-Way, 64 byte lines
    - 三级缓存 9 MB, 12-Way, 64 byte lines
    - 内存: 海力士 DDR4 2666MHz 8GB

实验方法和步骤

文件读写

- 考虑使用fsteam头文件下的ofstream以及ifstream流来轻松实现文件读写。
- 打印时利用`<iomanip`来格式化输出
- 需注意文件目录组织下，采用相对路径较为简单，个人因为使用Clion缘故，其运行时当前目录为cmake-build-debug,所以文件路径设置为

  `"..\\..\\input\\input11.txt"`等
- 此外，Clion不支持中文路径，需要注意
- 在使用vscode时，则需换成绝对路径。

计时

- 采用网上推荐的格式，在经过试验后，发现微秒级的计时即可让结果显示较为方便。
- 考虑采用<windows.h>头文件下一个us级的计时方式

```
//clock计时参数的声明
LARGE_INTEGER nFreq;
LARGE_INTEGER t1;
LARGE_INTEGER t2;
double dt;
```

```
QueryPerformanceFrequency(&nFreq);
QueryPerformanceCounter(&t1);
具体需要计时的操作;//矩阵链乘计算
QueryPerformanceCounter(&t2);
dt =(t2.QuadPart-t1.QuadPart)/(double)nFreq.QuadPart;
time=dt;
```

- 实验结果发现此时计时精度较高，每个数据规模下均有较为合理的测量结果。

Kruskal算法

数据结构

- 不相交的数据集合考虑用不相交集合森林来实现，并且只需要每个节点保留一个序号值即可对应到图中的点
- 图中的点也可以根据序号值来找到该树节点
- 不需要刻意存储边，直接把生成的所有边（一个结构体，包括左右端点的序号以及边权重）丢到小根堆里
- ```
  struct Node{
      Node* p;//父节点
      int rank;//高度
      int serial;//序号
  };

  struct Edge {
      int w;//权重
      int u;//左端点
      int v;//右端点
  };
  ```
- ```
  vector<Node> node_set(512);//存放分离数据集合的结点
  int edge_count[512][513];//用于判断该边是否存在及已经有的边数,纵列多出来的一列用于存放边数
  vector<Edge> Edge_heap;//存放生成的所有边
  vector<Edge> result;//存放Kruskal算法结果
  ```

generate

- 根据顶点数为对应无向图随机生成边
- 利用邻接矩阵的思想，用数组edge_count的0/1标记来指示两点间是否有边存在，并且数组多出一列记录该顶点已经生成的边数。注意无向图里，邻接矩阵的对称性需要考虑进来。
- 首先为某个顶点u随机生成边数，并与当前其还能加入的边数比较，看实际还要生成多少条边

```
num=1+rand()%(int(N/2));
int temp=(num <= int(N/2)-edge_count[i][N]) ? num:int(N/2)-edge_count[i][N];
```

- 实际生成每一条边时，首先随机生成边的另一端点v，若该边已存在，则重新生成端点v；若顶点v已生成的边数已有N/2,则放弃生成该条边。最后随机生成边的权值。

```
for(int j=0; j<temp ;j++){//依次生成每一条边的权重，同时判断该边所选择的另一个节点是否冲突
    int v=rand()%N;//该边的另一条顶点
    if(edge_count[i][v]){
        j--;
        continue;
    }
    else if(edge_count[v][N]>=int(N/2)) continue;
    else{
        int length=1+rand()%20;//随机生成边的权重
        Edge_heap.push_back(Edge{length,i,v});
        edge_count[i][N]++;
```

```
            edge_count[v][N]++;
            edge_count[i][v]=1;
            edge_count[v][i]=1;
        }
    }
```

分离数据集合的操作

- 与课本上算法基本一致，通过按秩合并以及路径压缩的思想来完成高效算法

Mst_Kruskal

- 本质是一个贪心算法
- 首先把每个顶点单独作为一个集合
- 通过C++提供的堆算法来实现最小优先队列（小根堆）
- 遍历所有边，每次从最小优先队列里取出权重最小的边，若边对应的两端点不在一个集合，则调用Union操作合并,并把对应的边加入result结果集合里。
- ```
  void Mst_Kruskal(int N){
      for(int i=0;i<N;i++){
          node_set[i].serial=i;
          Make_Set(&node_set[i]);
      }
      make_heap(Edge_heap.begin(),Edge_heap.end(),cmp);//生成最小优先队列
      while(Edge_heap.size()>0){
          pop_heap(Edge_heap.begin(),Edge_heap.end(),cmp);
          Edge l = Edge_heap.back();
          Edge_heap.pop_back();
          //cout<<l.w<<endl;
          if(Find_set(&node_set[l.u]) != Find_set(&node_set[l.v])){
              result.push_back(l);
              Union(&node_set[l.u],&node_set[l.v]);
          }
      }
  }
  ```

Johnson算法

- 注：算法的实现的正确性由main.cpp里的test函数检查（参考书上例子的数据构造）

数据结构

- 考虑使用邻接链表
  ```

- ```
  struct Edge {
      int w;//权重
      int u;//左端点编号
      int v;//右端点编号
      Edge* next;
  };
  struct Node{
      Edge* head;//指向第一条边
      int d;
      int pi;//前驱节点编号
      int h;//存储对应结点的h函数值
  };
  ```

  采用比较简单的结构体即可实现，各个属性的含义已经标示在代码里。

- ```
  //另外需要专门存放有向图中的各个数据，因为邻接链表只是一种形式(通过指针)，但实际的结点还是利用vector存放
  vector<Node> node_set(730);//存放邻接链表的结点集合
  int edge_count[729][729];//用于判断该边是否存在
  vector<Edge> Edge_set;//存放生成的所有边
  ```

- 需注意关于vector的一个坑：若不事先指定vector大小，那么会有一个默认size，当push进vector的元素容量超过时，会在内存中重新分配一个更大的空间，此时会导致本来的指针指向的地址无效的问题。所以采用声明变量时把其大小一并指明。

generate

- 根据顶点数N,以及对数底数m来为有向图中每个顶点随机生成边

- ```
  for(int i=0;i<N;i++){
      for(int j=0; j<num ;j++){//依次生成每一条边的权重，同时判断该边所选择的另一个节点是否冲突
          int v=rand()%N;//该边的另一个顶点
          if(edge_count[i][v]){
              j--;
              continue;
          }
          else{
              int length=rand()%51;//随机生成边的权重[0,50]
              Edge_set.push_back(Edge{length,i,v,NULL});
              edge_count[i][v]=1;
          }
      }
  }
  ```

  首先是随机生成边的另一个顶点，同时判断该节点是否冲突（即是否已经存在这条边，利用edge_count数组标记。若是，则重新生成），再生成每一条边的权重。

- 全部生成完毕后，采用尾插法构建邻接链表

Initialize_single_source

- Bellman-Ford和Dijkstra算法里的初始化操作，初始化d和pi值
- 其中pi初始化为-1，d初始化为INT_MAX

```
#define INT_MAX 50000//每条路径最多长 729*50 <50000
```
```

Relax

- 与书一致，不作过多阐述

Bellman_Ford

- 同课本，通过对每条边各进行N-1次松弛操作,逐渐降低从源结点s到每个结点v的最短路径的估计值v.d。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。
- ```
  bool Bellman_Ford(int s,int N){
      //s为源结点的编号,N为结点总数,算完之后，每个结点的d和pi为最短路径相关的值,O(VE)
      Initialize_single_source(s,N);
      for(int i=0;i<N-1;i++){
          for(auto iter=Edge_set.begin();iter!=Edge_set.end();iter++){
              Relax((*iter).u, (*iter).v, (*iter).w);
          }
      }
      for(auto iter=Edge_set.begin();iter!=Edge_set.end();iter++){
          int u=(*iter).u;
          int v=(*iter).v;
          if(node_set[v].d > node_set[u].d + (*iter).w){
              return false;
          }
      }
      return true;
  }
  ```

Dijkstra

- 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。
- 按课本所述，采用C++提供的优先队列来临时存放各个顶点的指针
- ```
  class cmp{
  public:
      bool operator() (Node *node1, Node *node2){
          return node1->d > node2->d;
      }
  };

  priority_queue<int,vector<Node*>,cmp> tmpNode_set;//最小优先队列
  ```

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
- 具体步骤（参考教材）：
  - 把G扩展为G'，加入结点s，s到原图中每个顶点都有一条权值为0的边

    ```
    for(int i=N-1;i>=0;i--){//初始化计算G'
        Edge_set.push_back(Edge{0,N,i,NULL});
        node_set[N].head=&Edge_set.back();
        node_set[N].head->next=temp;
        temp=node_set[N].head;
    }
    ```

- 在G'上调用 Bellman-Ford 算法
- 根据 Bellman-Ford 算法的结果设置h(v)

```
for(int i=0;i<N+1;i++){
    node_set[i].h=node_set[i].d;//初始化每个结点的h函数
}
```

- 重新计算权重函数

```
for(auto iter=Edge_set.begin();iter!=Edge_set.end();iter++){//重赋权重
    int u=(*iter).u;
    int v=(*iter).v;
    (*iter).w=(*iter).w + node_set[u].h - node_set[v].h;
}
```

- 对每个顶点u，调用Dijkstra函数计算u到其他所有结点的最短路径，并将结果按实验要求输出到文件里，注意到顶点u到顶点u自己的路径就不做考虑，也未输出到文件里。

```
for(int i=0;i<N;i++){
    Dijkstra(i,N+1);
    for(int j=0;j<N;j++){
        if(i!=j){
            if(print_path(i,j,f)){
                f<<"  "<<node_set[j].d + node_set[j].h -node_set[i].h<<"
)"<<"\n";
            }
            else{//两节点间不存在最短路径
                f<<"no pass for node"<<i<<","<<j<<"\n";
            }
        }
    }
}
```

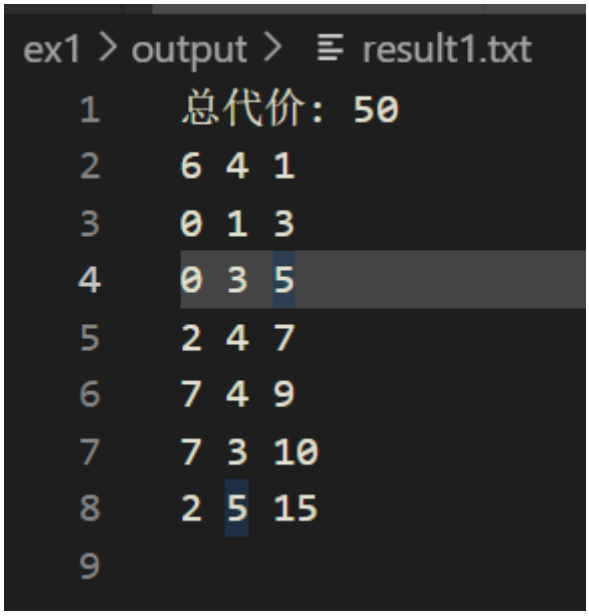
print_path

- 递归将结果输出到result文件里，若两顶点间存在路径，则返回true，否则返回false
- ```
  bool print_path(int u,int v,ofstream &f){//输出从u到v的最短路径上的所有结点
      if(u!=v ){
          if(node_set[v].pi==-1) return false;
          else{
              print_path(u,node_set[v].pi,f);
              f<<v<<",";
          }
      }
      else {
          f<<"( "<<v<<",";
      }
      return true;
  }
  ```

实验结果与分析
  ```

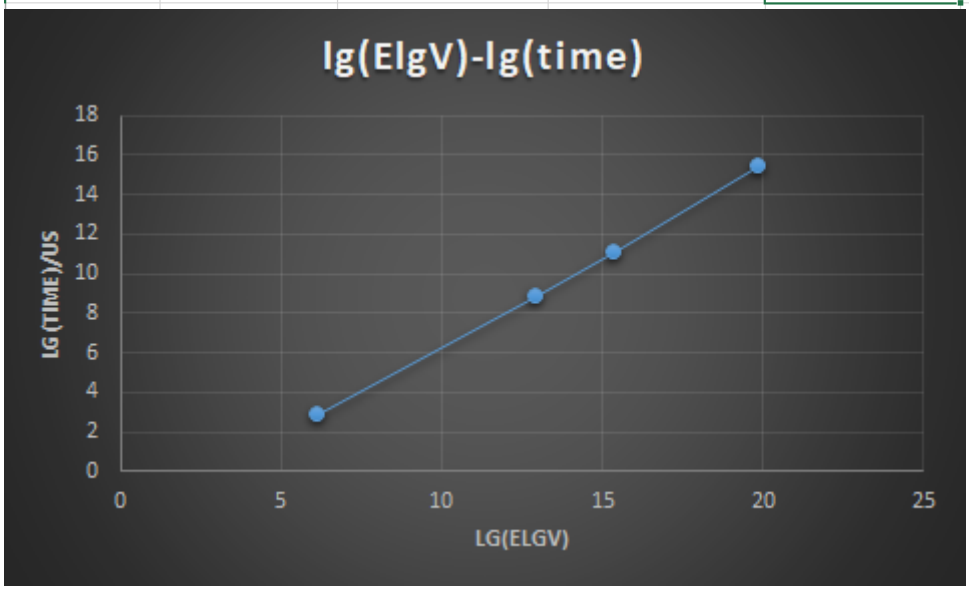
Kruskal算法

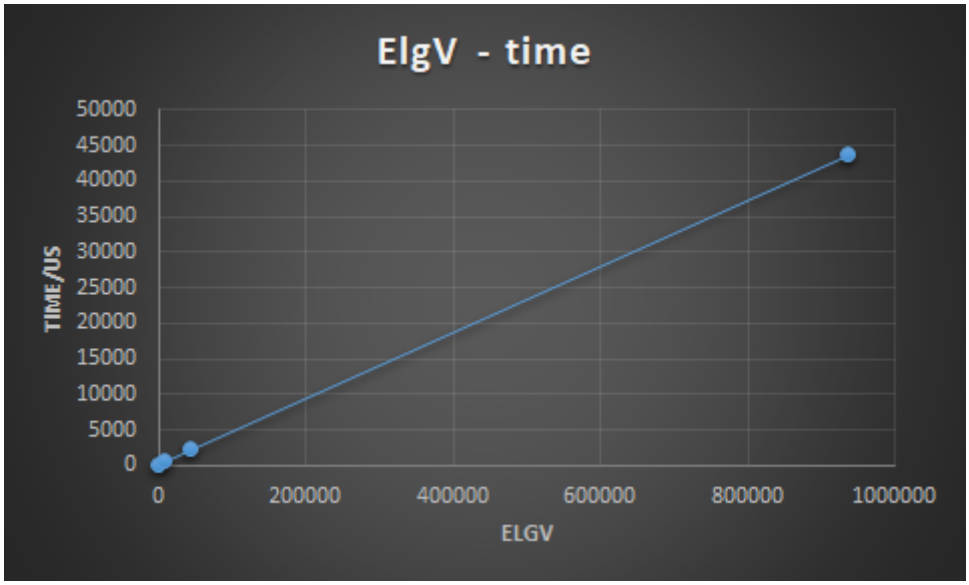
- result1.txt文件示例：



- 时间复杂度分析：因为使用了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)

顶点数V	边数E	ElgV	lg(ElgV)	time(us)	lg(time) /us
8	23	69	6.108524457	7.4	2.887525271
64	1279	7674	12.90576305	467.1	8.867587635
128	6140	42980	15.39137786	2195	11.10000522
512	104288	938592	19.84013864	43559.7	15.4107064

- 

- 

- 由图像分析可知，实际时间复杂度与理论时间复杂度吻合。

Johnson算法

- 输出的result文件示例，其中存在路径时，括号内最后一个数即为路径长度。

  no pass for node 0,1 表示从顶点0到顶点1不存在路径。

```
no pass for node0,1
no pass for node0,2
( 0,19,13,18,7,3,  110 )
no pass for node0,4
no pass for node0,5
( 0,19,15,6,  55 )
( 0,19,13,18,7,  85 )
no pass for node0,8
no pass for node0,9
no pass for node0,10
( 0,19,13,18,7,11,  109 )
```

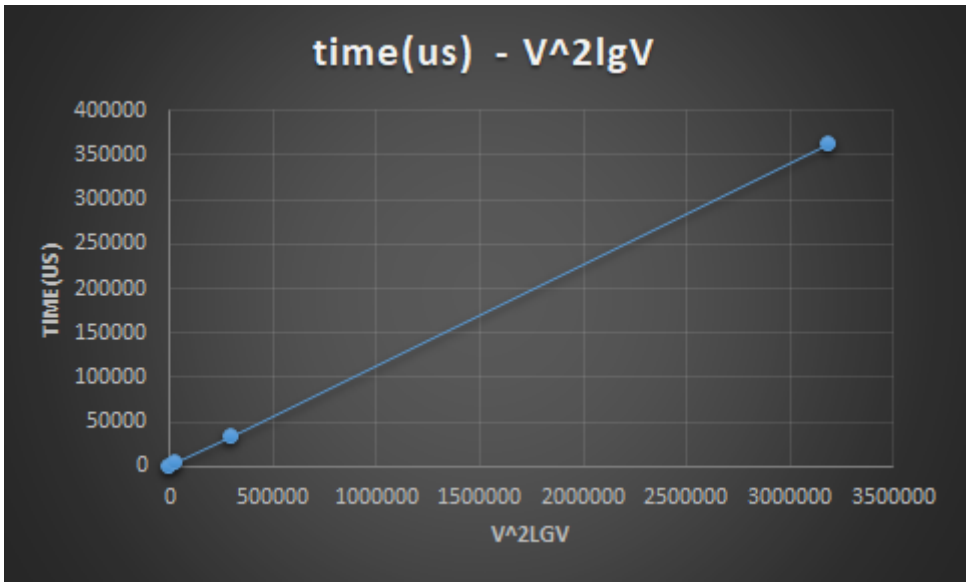
- 输出的time.txt示例，为方便比较，从上到下对应的result文件为11，21，31，41，12，22，32，42

ex2 > output > time.txt

```
384.9
3423.7
34672.7
362982
342.4
3400
32085.4
335345
```

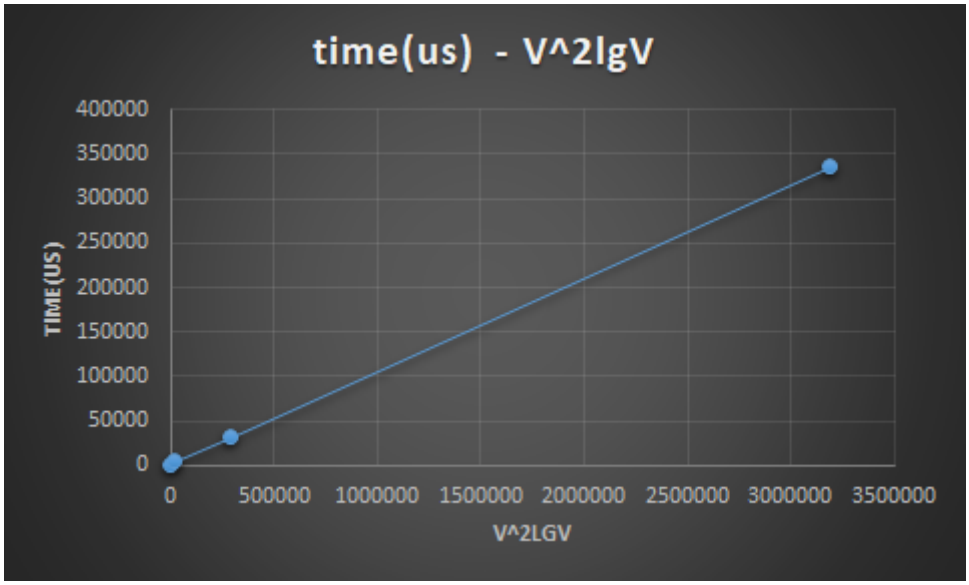
- 由于 Johnson 算法调用的 Dijkstra 算法部分使用了 C++ 中的优先队列来实现，由于优先队列本质上是一个堆，不论其是二叉堆还是斐波那契堆，由于实验中 E=kV ，理论时间复杂度均为 $O(V^2lgV)$
- 另外注意到，print_path函数递归调用不是算法本身要求，所以在测量时间时把Johnson函数里对应的输出部分注释掉了。
- 时间复杂度分析
  - log5N

V	lgV	E	V^2 lgV	time(us)
27	3	54	2187	384.9
81	4	162	26244	3423.7
243	5	729	295245	34672.7
729	6	2916	3188646	362982



  - log7N

V	lgV	E	V^2 lgV	time(us)
27	3	27	2187	342.4
81	4	162	26244	3400
243	5	486	295245	32085.4
729	6	2187	3188646	335345



- 可以看到，无论是log5N还是log7N的图像，结果均为一条直线，即实际时间复杂度与理论时间复杂度一致。

实验总结

- 通过本次实验，我对课上所学的最小生成树相关算法有了更深刻的认识，并且实现了分离数据集合的代码，从而降低了算法的时间复杂度。
- 通过实现johnson算法，对于单源最短路径、所有结点对的最短路径这两个问题有了更清晰的认知，无论是Bellman-ford，Dijkstra算法，还是里面用到的relax思想、C++优先队列，都算是从无到有的学习过程。此外，johnson算法里的重赋权重思想也十分重要，值得实现并构造样例来理解。
- 实验里遇到了一些意料之外的bug，debug耗费了大量的时间，也给自己带来一些启发
    - vector分配空间机制
    - 文件路径(string)的字符串乱码问题
    - 对于复杂的实现，个人认为应该把文件划分为小的模块，首先应把每个模块作好充分测试，这样合并起来的时候会有更好的保障而不会导致找不到错误位置而抓狂。