图算法实验报告

实验内容与要求

Kruskal算法

内容：

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。

要求:

- 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。
- 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。
- 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。

Johnson算法

内容:

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。)

要求

- 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(log5N、log7N));
- 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[-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。

实验设备和环境

- 硬件条件:一台 PC 机，
- 软件条件: mac 操作系统， VS code 编辑器， gcc 编译器 。

实验方法与步骤

1. 构建文件目录

   建立一个根文件夹实验需建立根文件夹，文件夹名称为:编号-姓 名-学号-project4，在根文 件夹下需包括实验报告和 ex1 子文件夹， 和 ex2 文件夹。实验报告命名为编号-姓名-学号-project4.pdf， ex1子文件夹和 ex2 子文件夹又包含 3 个子文件夹:

   `input`文件夹:存放输入数据

   `src` 文件夹:源程序

   `output` 文件夹:输出数据

2. Kruskal算法
   - 先写一个按照指定要求 生成随机数的程序 作为图的基本信息。 事实上，这部分在本次实验中也占了比较重要的地位，因为对随机数的限制要求比较多， 且大多数是动态的限制 。但由于不是考察重点，不再赘述。
   - 然后在实现kruskal 算法前 先实现分离集合数据结构 。实现以下函数
     - `make_set`

       ```
       //不想交集合森林 make_set
       void make_set(int x){
           p[x] = x ;
           rk[x] = 0;
       }
       ```

     - `find_set`

       ```
       //find_set
       int find_set(int x){
            if(x != p[x]){
                p[x] = find_set(p[x]) ;
            }
            return p[x];
       }
       ```

     - `union_set`

```
//union_set
void link(int x,int y){
    if(rk[x] > rk[y]){
        p[y] = x;
    }else
    {
        p[x] = y;
        if(rk[x] == rk[y])
            rk[y] = rk[y] +1;
    }
}
void union_set(int x,int y){
    link(find_set(x) , find_set(y) );
}
```

- 一切准备就绪，就可以实现`kruskal` 算法了。该算法的结构大概如下：

  ```
  //kruskal 算法
      for( i = 0;i < edge_num ; i++){
          make_set(i);
      }
      sort(edge_list,edge_list+edge_num,MyCompare);  //排序
      for(i = 0 ; i < edge_num ; i++){  //把最小的边 合并
           if(find_set(edge_list[i].u) != find_set(edge_list[i].v)){
           MST_edge.push_back(edge_list[i]);
           union_set(edge_list[i].u , edge_list[i].v) ;
           }
      }
  ```

- 最后在主函数中 对不同规模的问题 运行`kruskal` 算法 。进行时间记录，以及写输出结果。

3. Johnson算法

- 还是 先写一个按照指定要求 生成随机数的程序 作为图的基本信息。不过这次较容易实现 。因为每个顶点有确定的边数 。
- 先实现`Bellman-Ford` 算法 。利用`Bellman-Ford`算法 求带有负边的 最短路径问题，来求h(x)。当有负环存在时删除负环的一条边 。

  算法如下：

  ```
      ININIAL_S(G,G->vex_num-1); //初始化 ,
      for( i = 0 ; i < G->vex_num - 1;i++){  //vex_num -1 次 轮 遍历
          for(it = G->edge_list.begin() ; it!=G->edge_list.end() ; it++)
  //对每条边遍历。
              Relax(&G->vex[it->u] , &G->vex[it->v] ,it->weight);
  //relax操作
      }
  ```

- 然后就可以求 `w'` ，是的每条边的权值都为正， 且原图的最短路径也是， 还是最短路径。
- 对新图进行 `Dijkstra` 算法求单源最短路径。这样对每个结点求单源最短路径，就得到了所有节点对的最短路径。Dijkstra的 算法实现如下：其中最小堆的实现是自己实现的 。

```
ININIAL_S(G , s) ; //初始化
//自己实现最小优先队列。
for(int i=0 ; i < G->vex_num ;i++){
    V[i] = i;
}
Build_Heap(G,V , V_size);  //建立最小堆
while(V_size != 0){
   u = Extract_min(G,V , V_size);
   V_size-- ; //得到最小的
   // 迭代s 输出最小距离
   for(v = G->vex[u].adj.begin();v != G->vex[u].adj.end() ;v++){
         Relax(&G->vex[u] , &G->vex[*v] , G->weight[u][*v]);
   }// 此时 u.d v.d 是在变的。所以最小堆已经不是最小堆了。
   Build_Heap(G,V,V_size); //重新维护最小堆性质。
}
```

实验结果与分析

1. 先展示`kruskal` 算法的结果。以小规模为例。

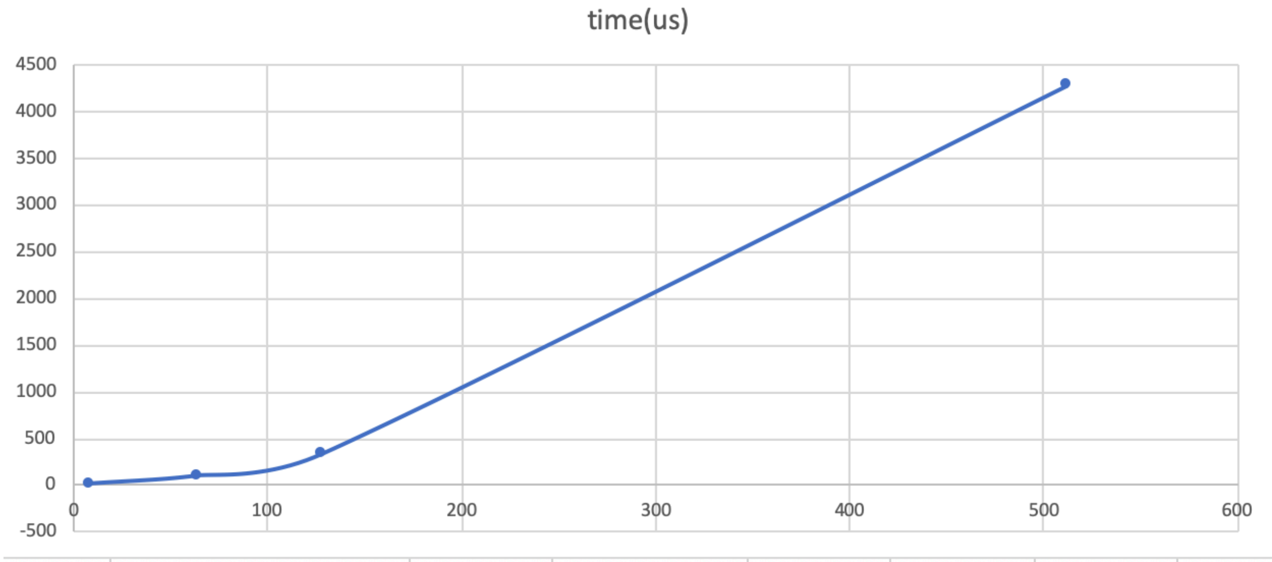
```
ex1 > input > input1.txt
0 5 14
0 7 1
0 6 16
0 3 19
1 5 15
1 2 19
3 5 14
3 6 14
4 7 15
4 6 3
4 5 2
4 2 16
```

```
ex1 > output > result1.txt
0 7 1
4 5 2
4 6 3
0 5 14
3 5 14
1 5 15
4 2 16
```

输入的图 **最小生成树**

```
ex1 > output > time.txt
The 8 scale problem of kruskal get the MST costs 0.000011 s ...
The 64 scale problem of kruskal get the MST costs 0.000098 s ...
The 128 scale problem of kruskal get the MST costs 0.000328 s ...
The 512 scale problem of kruskal get the MST costs 0.004281 s ...
```

不同规模运行的时间



各种规模的运行时间区线

分析： `kruskal` 算法的时间复杂度是 $O(E^2 + V + E)$ 。排序的时候使用了快排。而根据图中曲线也可以看出 时间曲线增长较快，基本符合预期的增长趋势。

2. 再展示`Johnson` 算法的结果。以小规模为例。

ex2 > input > input12.txt

```
0 13 26
1 26 16
2 6 28
3 20 47
4 0 1
5 23 17
6 21 5
7 12 48
8 9 -3
9 23 35
10 1 33
11 4 39
12 1 45
13 24 20
14 11 48
15 12 29
16 22 41
17 9 50
18 26 16
19 6 34
20 5 26
21 4 47
22 12 33
23 25 13
24 26 42
25 7 10
26 24 23
```

ex2 > output > result12.txt

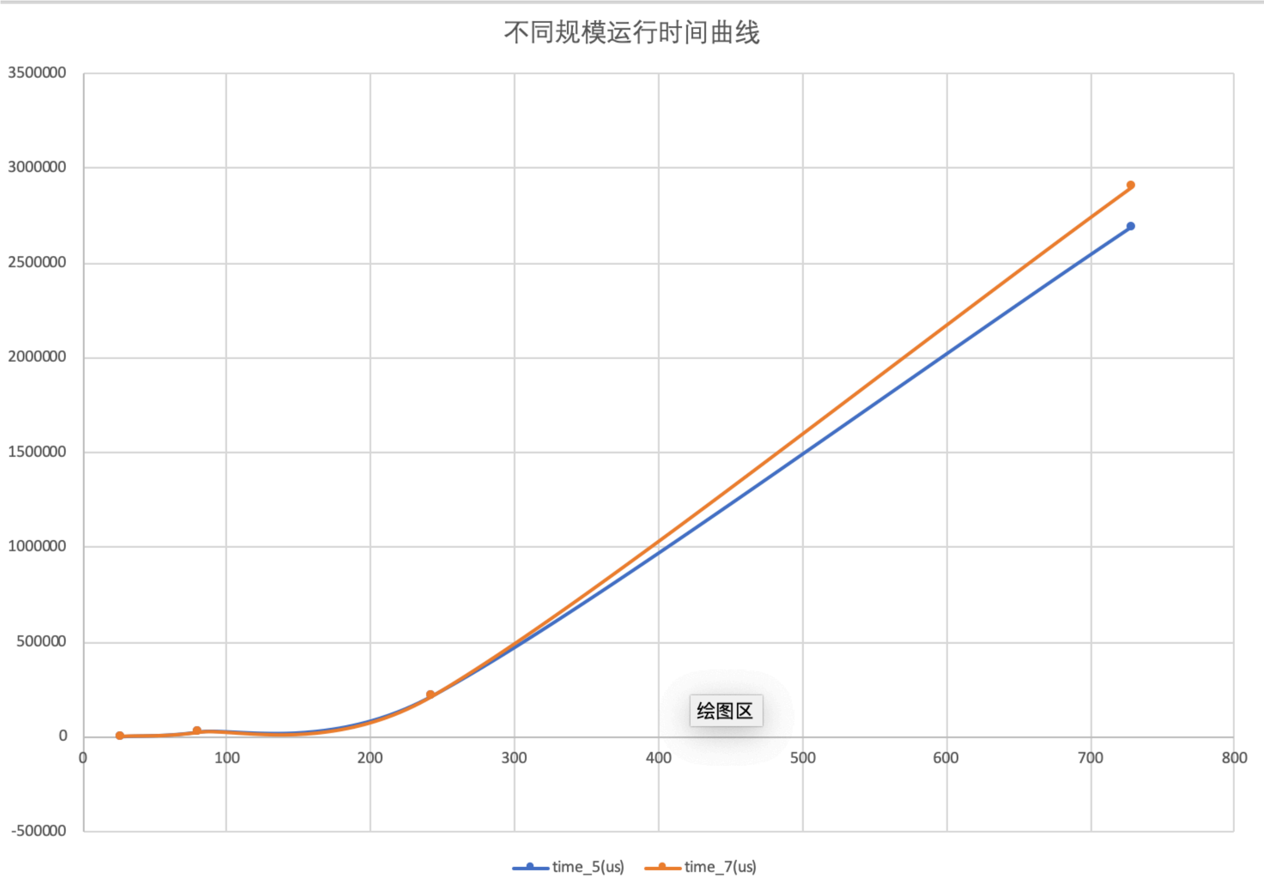
```
(0  1):there is no shortest path.
(0  2):there is no shortest path.
(0  3):there is no shortest path.
(0  4):there is no shortest path.
(0  5):there is no shortest path.
(0  6):there is no shortest path.
(0  7):there is no shortest path.
(0  8):there is no shortest path.
(0  9):there is no shortest path.
```

```
(0  10):there is no shortest path.
(0  11):there is no shortest path.
(0  12):there is no shortest path.
(0,13 26)
(0  14):there is no shortest path.
(0  15):there is no shortest path.
(0  16):there is no shortest path.
(0  17):there is no shortest path.
(0  18):there is no shortest path.
(0  19):there is no shortest path.
(0  20):there is no shortest path.
(0  21):there is no shortest path.
(0  22):there is no shortest path.
(0  23):there is no shortest path.
(0,13,24 46)
(0  25):there is no shortest path.
(0,13,24,26 88)
(1  0):there is no shortest path.
```

27节点输入 最短路径输出

```
ex2 > output > time.txt
The 27 scale series 0 problem   costs 0.002627 s ...
The 27 scale series 1 problem   costs 0.001703 s ...
The 81 scale series 0 problem   costs 0.027161 s ...
The 81 scale series 1 problem   costs 0.028341 s ...
The 243 scale series 0 problem   costs 0.271601 s ...
The 243 scale series 1 problem   costs 0.267916 s ...
The 729 scale series 0 problem   costs 2.904915 s ...
The 729 scale series 1 problem   costs 3.264240 s ...
```

各个规模的运行时间



不同规模 运行时间 横向纵向对比图

分析： 由于输入保证无负环， 所以在运行Johnson算法时只判断了一次有无负环。与之前的对负环删除一条边的操作相比较，时间复杂度减少为O(VE) 。如果输入有负环，则需要加入一层删除负环中的边的算法 。而这一段是相当耗时间的，因为需要不断的运行 Bellman-Ford算法。最坏情况下可以有O(E)条边都需要删除，此时算法的时间复杂度为O(E*VE) 。

实验总结

`kruskal` 算法较为简单，关键在于不相交集合 数据结构的实现，这里可以参照书上的路径压缩的实现方法。实验过程中也没有遇到什么问题。问题都出现在Johnson算法中 。 由于实验的要求 没有保证 输入中不含负环。所以在`Johnson` 算法中加入了一层对负环的处理 。处理方法是 删除掉负环中的一条边。注意删除的边可能是正边，也可能是负边，目的是拆环。这样以来算法的时间复杂度就变高了，而Johnson处理的图也不是原来输入的图了 。由于处理负环的方法不一样，导致同样的输入会有不同的输出。 好在最终要求输入边的权重是[0,50]。