实验四

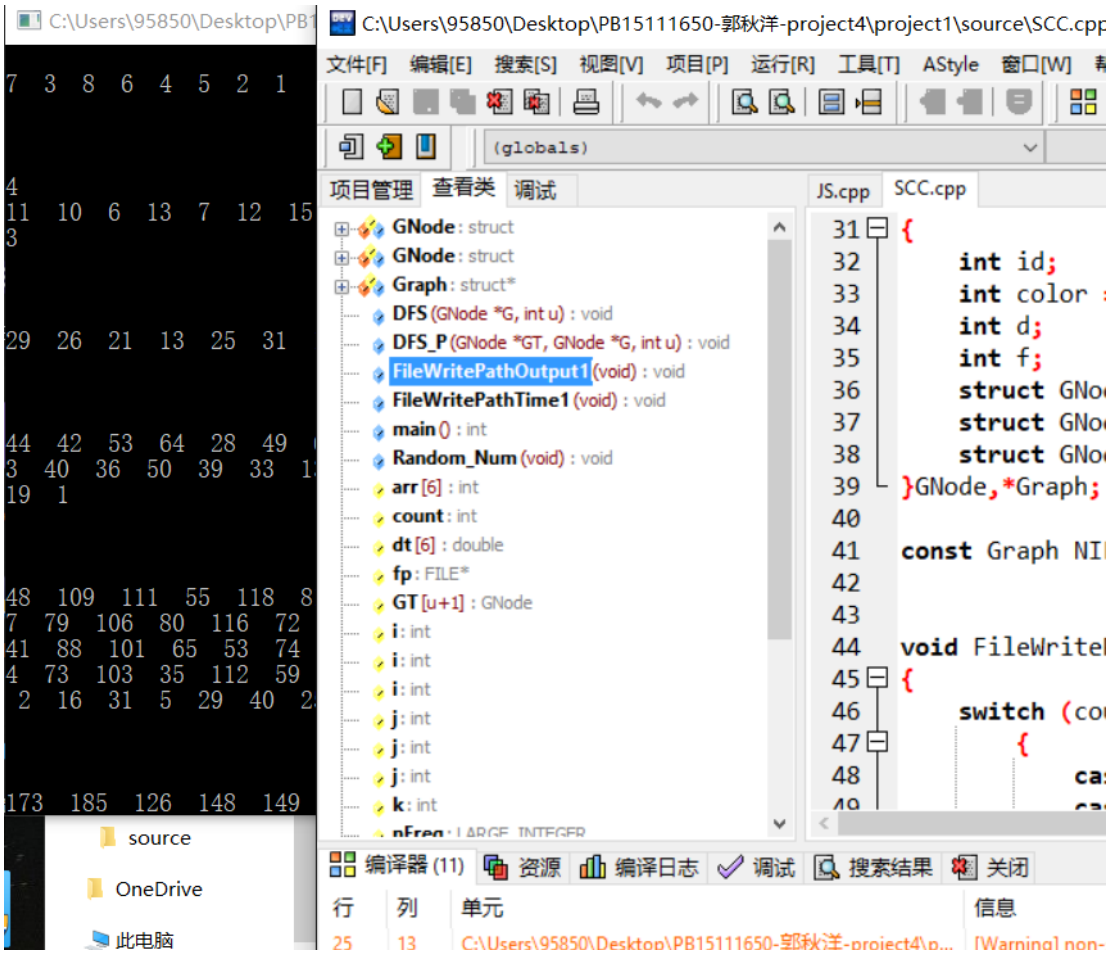
1. 实验要求

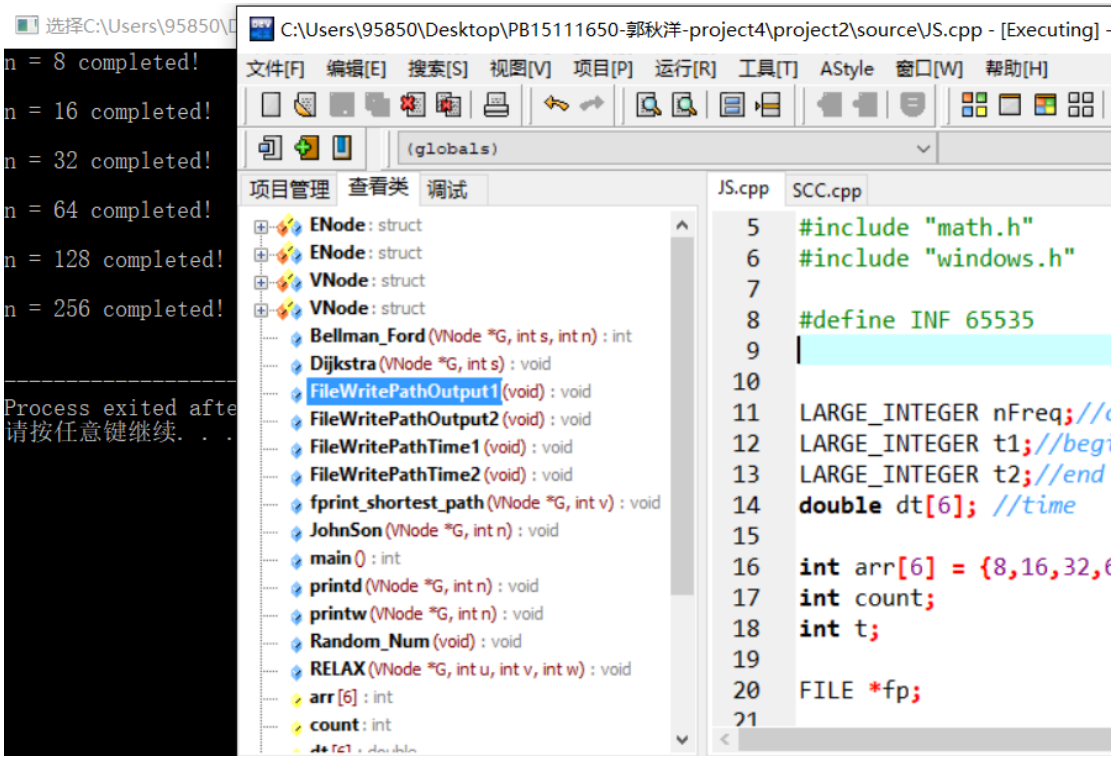
1)实现求有向图的强连通分量的算法。有向图的顶点数 N 的取值分别为：8、16、32、64、128、256，弧的数目为 NlogN，随机生成 NlogN 条弧，统计算法所需运行时间 ，画出时间曲线。
2)实现求所有点对最短路径的 Johnson 算法。生成连通的无向图，图的顶点数 N 的取值分别为： 8、16、32、64、128、256 ，边的数目为 NlogN，随机生成 NlogN 条边，统计算法所需运行时间 ，画出时间曲线。

2. 实验环境

编译环境：Dev C++ 5.11
机器内存：16G
时钟主频：2.20GHz

3. 实验过程





4. 实验关键代码截图（结合文字说明）

实验 1：

1. 全局变量：

   t 记录 DFS 中对结点操作的时间戳。

```
#define INF 65535


LARGE_INTEGER nFreq;//cpu frequency
LARGE_INTEGER t1;//begin
LARGE_INTEGER t2;//end
double dt[6]; //time

int arr[6] = {8,16,32,64,128,256};
int count;
int t;

FILE *fp;
```

2. 图的“邻接链表”

```
enum color
{
    WHITE,
    GRAY,
    BLACK
};

typedef struct GNode
{
    int id;
    int color = WHITE;
    int d;
    int f;
    struct GNode *Node = NULL;
    struct GNode *parent = NULL;
    struct GNode *next = NULL;
}GNode,*Graph;

const Graph NIL = new GNode;
```

由于开始对邻接链表不熟，所以误以为链表里存放的是结点。所以没有定义边结构，而是全部用结点结构表示，并且额外定义了 id 标识结点，Node 指向本结点，这两个属性是多余的。所以这不是标准的邻接链表，但仍然可以实现各种图的操作。d 和 f 为时间戳，分别记录结点第一次被发现时（涂上灰色）和完成对结点邻接链表扫描时（涂上黑色）的时间 t。

3. 图的生成函数

```
while(1)
{
    int r1 = 1+rand()%n;
    int r2 = 1+rand()%n;
    if(e[r1][r2] != 1)
    {
        e[r1][r2] = 1;
        fprintf(fp,"%d ",r1);
        fprintf(fp,"%d\n",r2);
        k++;
    }
    if(k == m)
        break;
}
if(k < m)
    printf("less than nlogn!");
```

这里使用邻接矩阵生成图更方便。

4. DFS 算法

```
DFS_VISIT(GNode *G,int u)
{
    int i,j;
    GNode *p;
    ++t;
    G[u].d = t;
    G[u].color = GRAY;
//  printf("%d\n",G[u].f);
    p = G[u].next;
    while(p)
    {
        if(p->Node->color == WHITE)
        {
            p->parent = &G[u];
            p->Node->color = GRAY;
            DFS_VISIT(G,p->Node->id);
        }
        p = p->next;
    }
    G[u].color = BLACK;
//  printf("%d  ",u);
    t = t + 1;
    G[u].f = t;
//  printf("%d\n",G[u].f);
}
```

```
void DFS(GNode *G,int u)
{
    int i,v;
    t = 0;
    for(i = 1;i <= u;i++)
    {
        if(G[i].color == WHITE)
        DFS_VISIT(G,i);
    }

}
```

这里 p->Node->id 可直接用 u 代替，p->Node->color 可直接用 G[u].color 代替。

5 读取文件中的图

这里用了一个邻接矩阵存储读出来的图主要是为了测试时方便打印图矩阵以及方便图转置，也可直接生成邻接链表。

```
    while(fgets(line,31,fp))
    {
        int i = 0;
        int num = 0;
        while (line[i] >= '0' && line[i] <= '9')
        {
            num = num * 10 + line[i] - '0';
            ++i;
        }

        e1[k] = num;

        ++i;
        num = 0;
        while (line[i] >= '0' && line[i] <= '9')
        {
            num = num * 10 + line[i] - '0';
            ++i;
        }

        e2[k] = num;


        e[e1[k]][e2[k]] = 1;
```

6 生成邻接链表

```
Graph v = new GNode;
G[e2[k]].id = e2[k];
G[e2[k]].Node = &G[e2[k]];
v->Node = &G[e2[k]];
if(!G[e1[k]].next)
{
    G[e1[k]].id = e1[k];
    G[e1[k]].Node = &G[e1[k]];
    G[e1[k]].next = v;
    p[e1[k]] = v;
}
else
{
    p[e1[k]]->next = v;
    p[e1[k]] = p[e1[k]]->next;
}
k++;
if(k == m + 1)
    break;
```

7 图转置

```
    for(i = 1;i <= u;i++)
        {
            for(j = i;j <= u;j++)
            {
                if(*(e + i*(u+1) + j) == 1 && *(e + j*(u+1) + i) == 0)
                {
                    *(e + i*(u+1) + j) = 0;
                    *(e + j*(u+1) + i) = 1;
                }
                else if(*(e + i*(u+1) + j) == 0 && *(e + j*(u+1) + i) == 1)
                {
                    *(e + i*(u+1) + j) = 1;
                    *(e + j*(u+1) + i) = 0;
                }

            }
        }
```

邻接矩阵进行图转置很方便，然后再按 6 的方法生成邻接链表。

8. 按第一次 DFS 生成的 u.f 从大到小对 GT 运行 DFS_VIDSIT_P

```
void DFS_P(GNode *GT,GNode *G,int u)
{
    int i,j,min,temp,sort[u+1];
    t = 0;
    for(i = 1;i <= u;i++)
        sort[i] = G[i].f;
    for(i = 1;i <= u - 1;i++)
    {
        min = i;
        for(j = i + 1;j <= u;j++)
        {
            if(sort[min] >= sort[j])
            {
                min = j;
            }
        }
        temp = sort[i];
        sort[i] = sort[min];
        sort[min] = temp;
    }                           //sort
    for(i = u;i >= 1;i--)
    {
        j = 1;
        while(j)
        {
            if(G[j].f== sort[i])
                break;
            j++;
        }
        if(GT[j].color == WHITE)
        {
            QueryPerformanceCounter(&t1);
            DFS_VISIT_P(GT,j);
            QueryPerformanceCounter(&t2);
            printf("\n");
            FileWritePathOutput1();
            fprintf(fp,"\n");
```

9 在 DFS_VISIT_P 中打印连通分量

```
DFS_VISIT_P(GNode *G,int u)
{
    int i,j;
    GNode *p;
    ++t;
    G[u].d = t;
    G[u].color = GRAY;
//  printf("%d\n",G[u].f);
    p = G[u].next;
    while(p)
    {
        if(p->Node->color == WHITE)
        {
            p->parent = &G[u];
            p->Node->color = GRAY;
            DFS_VISIT_P(G,p->Node->id);
        }
        p = p->next;
    }
    G[u].color = BLACK;
    printf("%d  ",u);
    FileWritePathOutput1();
    fprintf(fp,"%d ",u);
    fclose(fp);
    t = t + 1;
    G[u].f = t;
//  printf("%d\n",G[u].f);
}
```

10 在 DFS_P 中打印求解每个连通分量的运行时间

```
        FileWritePathTime1();
        fprintf(fp,"%f\n",(t2.QuadPart - t1.QuadPart)/ (double)nFreq.QuadPart *1000000);
        fclose(fp);
```

实验 2：

1 全局变量：

```
#define INF 65535


LARGE_INTEGER nFreq;//cpu frequency
LARGE_INTEGER t1;//begin
LARGE_INTEGER t2;//end
double dt[6]; //time

int arr[6] = {8,16,32,64,128,256};
int count;

FILE *fp;
```

2 邻接链表

```
typedef struct ENode
{
    int u = 0;
    int v = 0;
    ENode *next = NULL;
    int w = INF;
}ENode;

typedef struct VNode
{
    int id = 0;
    ENode *first = NULL;
    int key = INF;

    int parent = 0;
    int d = INF;
    int indegree = 0;
}VNode;
```

分别定义结点结构 VNode 和边结构 ENode。ENode 中，属性 u，v 为边的两端，方向 u->v，w 为权值；VNode 中 id 标识结点，key 为结点携带的值，d 为最短路径上界，indegree 为入度。本实验中 id，key，indegree 没有用到。

3 图的生成函数

```
        while(1)
        {
            int r1 = 1+rand()%n;
            int r2 = 1+rand()%n;
            if(G[r1][r2] == INF)
            {
                if(rand()%n == 1)
                {
                    G[r1][r2] = - (1 + rand()%((int)log2(n)));
                }
                else
                    G[r1][r2] = 1 + rand()%n;

                fprintf(fp,"%d ",r1);
                fprintf(fp,"%d ",r2);
                fprintf(fp,"%d\n",G[r1][r2]);
                k++;
            }
            if(k == m)
                break;
        }
        if(k < m)
            printf("less than nlogn!");
        fclose(fp);
```

生成的图满足：1/n 的边的权值为负的，且绝对值不大于 log n。其他边权值为正，且不大于 n。同样用邻接矩阵存储。

4 读取文件中的图

```
while(fgets(line,31,fp))
{
    int i = 0;
    int Nflag = 0;
    int num = 0;
    while (line[i] >= '0' && line[i] <= '9')
    {
        num = num * 10 + line[i] - '0';
        ++i;
    }
    v1[k] = num;

    ++i;
    num = 0;
    while (line[i] >= '0' && line[i] <= '9')
    {
        num = num * 10 + line[i] - '0';
        ++i;
    }
    v2[k] = num;

    ++i;
    num = 0;
    if(line[i] == '-')
    {
        ++i;
        Nflag = 1;
    }
    while (line[i] >= '0' && line[i] <= '9')
    {
        num = num * 10 + line[i] - '0';
        ++i;
    }
    if(Nflag == 1)
        G[v1[k]][v2[k]] = -num;
        else
        G[v1[k]][v2[k]] = num;
```

同样是先读到邻接矩阵中方便打印。在实验二由于不需要转置所以可以直接存到邻接链表中。

5 存储到邻接链表

```
    ENode *e = new ENode;
    e->w = G[v1[k]][v2[k]];
    e->u = v1[k];
    e->v = v2[k];

    if(!V[v1[k]].first)
    {
        V[v1[k]].id = v1[k];
        V[v1[k]].first = e;
        V[v1[k]].indegree++;
        p[v1[k]] = e;
    }
    else
    {
        p[v1[k]]->next = e;
        p[v1[k]] = p[v1[k]]->next;
    }
```

6 初始化和 RELAX 函数

```
INITIALZE_SINGLE_SOURCE(VNode *G,int s,int n)
{
    for(int i = 1;i <= n;i++)
    {
        G[i].d = INF;
        G[i].parent = 0;
    }
    G[s].d = 0;
}

void RELAX(VNode *G,int u,int v,int w)
{
    if(G[v].d > (G[u].d + w))
    {
        G[v].d = G[u].d + w;
        G[v].parent = u;
    }
}
```

7 Bellman_Ford 算法

```
int Bellman_Ford(VNode *G,int s,int n)
{
    INITIALZE_SINGLE_SOURCE(G,s,n);
    for(int i = 1;i < n;i++)
    {
        for(int j = 1;j <= n;j++)
        {
            ENode *q = G[j].first;
            while(q)
            {
                RELAX(G,q->u,q->v,q->w);
                q = q -> next;
            }
        }
    }
    for(int i = 1;i <= n;i++)
    {
        ENode *q = G[i].first;
        while(q)
        {
            if(G[q->v].d > (G[q->u].d + q->w))
                return 0;
            q = q -> next;
        }
    }
    return 1;
}
```

8 Dijkstra 算法

```
void Dijkstra(VNode *G,int s)
{
    int n = arr[count];
    INITIALZE_SINGLE_SOURCE(G,s,n);
    int u;
    int *Q[n+1];
    memset(Q,INF,sizeof(Q));
    for(int i = 1;i <= n;i++)
        Q[i] = &G[i].d;
    while(n)
    {
        int min = INF;
        for(int i = 1;i <= arr[count];i++)
        {
            if(Q[i] == NULL)
                continue;
            if(*Q[i] < min)
            {
                min = *Q[i];
                u = i;
            }
        }
        Q[u] = NULL;
        n--;
        ENode *q = G[u].first;
        while(q)
        {
            RELAX(G,q->u,q->v,q->w);
            q = q->next;
        }
    }
}
```

由于数据规模较小，所以这里使用简单的线性数组作为优先队列。

9 JohnSon 算法

在结点数组最后增加 s 结点，增加 n 条指向 n 个结点的边，权值均为 0，对 s 调用 Bellman_Ford 算法。

```
void JohnSon(VNode *G,int n)
{
    QueryPerformanceCounter(&t1);
    int h[n+1];
    memset(h,0,sizeof(h));
    int D[n+1][n+1];
    for(int i = 1;i <= n;i++)
    {
        for(int j = 1;j <= n;j++)
            D[i][j] = INF;
    }
    int s = n + 1;
    G[s].d = 0;
    ENode E[n+1];
    G[s].first = &E[1];
    for(int i = 1;i <= n;i++)
    {
        E[i].u = n+1;
        E[i].v = i;
        E[i].w = 0;
        if(i == n)
            break;
        E[i].next = &E[i+1];
    }
    if(!(Bellman_Ford(G,s,n+1)))
        printf("the input graph contains a negative-weight cycle\n");
    else
```

若无负权值圈则计算每个 h[i],然后重新计算边权值。最后对除 s 的每个结点调用 Dijkstra 算法，每次调用均根据 parent 指针递归打印最短路径经过的结点，再打印最短路径值。找到所有点对最短路径后打印运行时间。

```
void fprint_shortest_path(VNode *G,int v)
{
    if(v == 0)
        return;
    fprint_shortest_path(G,G[v].parent);
    fprintf(fp,"%d ",v);
}
```

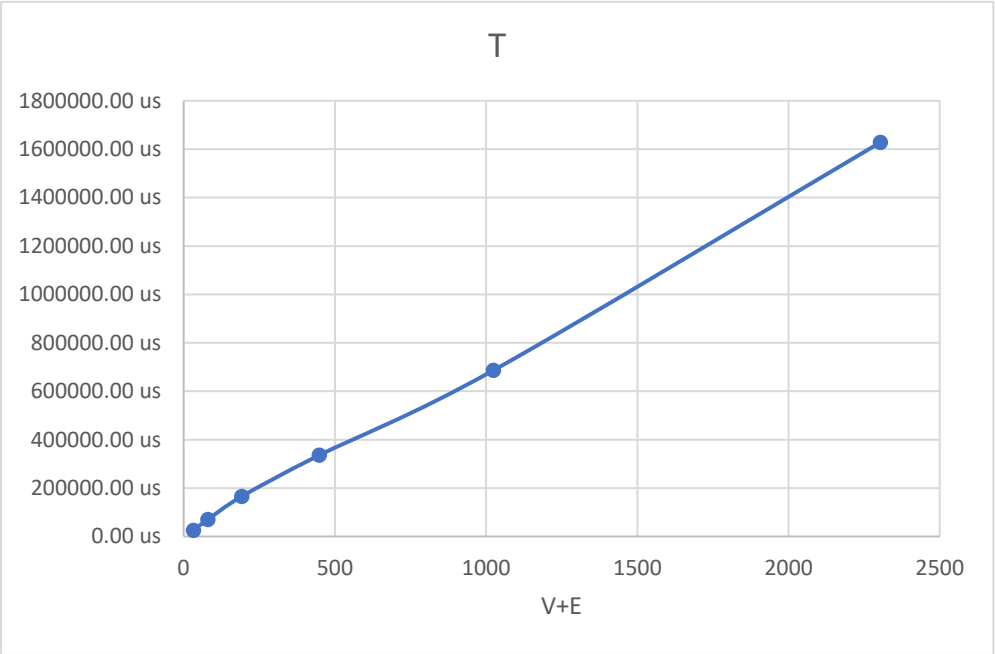
```
    fclose(fp);
    QueryPerformanceCounter(&t2);
    FileWritePathTime2();
    fprintf(fp,"%f\n",(t2.QuadPart - t1.QuadPart)/ (double)nFreq.QuadPart *1000000);
    fclose(fp);
```

```
for(int i = 1;i <= n;i++)
    h[i] = G[i].d;
for(int j = 1;j <= n;j++)
{
    ENode *q = G[j].first;
    while(q)
    {
        q->w = q->w + h[q->u] - h[q->v];;
        q = q -> next;
    }
}
//  printd(G,n);
//  printw(G,n);
FileWritePathOutput2();
for(int i = 1;i <= n;i++)
{
    Dijkstra(G,i);
//  printd(G,n);
//  printw(G,n);
    for(int k = 1;k <= n;k++)
    {
        D[i][k] = G[k].d + h[k] - h[i];
        if(D[i][k] == INF)
        {
            fprintf(fp,"%d ",i);
            fprintf(fp,"%d ",k);
            fprintf(fp,"Length: %c\n",'$');
        }
        else
        {
            fprint_shortest_path(G,k);
            fprintf(fp,"Length: %2d\n",D[i][k]);
        }
    }
```

实验结果、分析（结合相关数据图表分析）

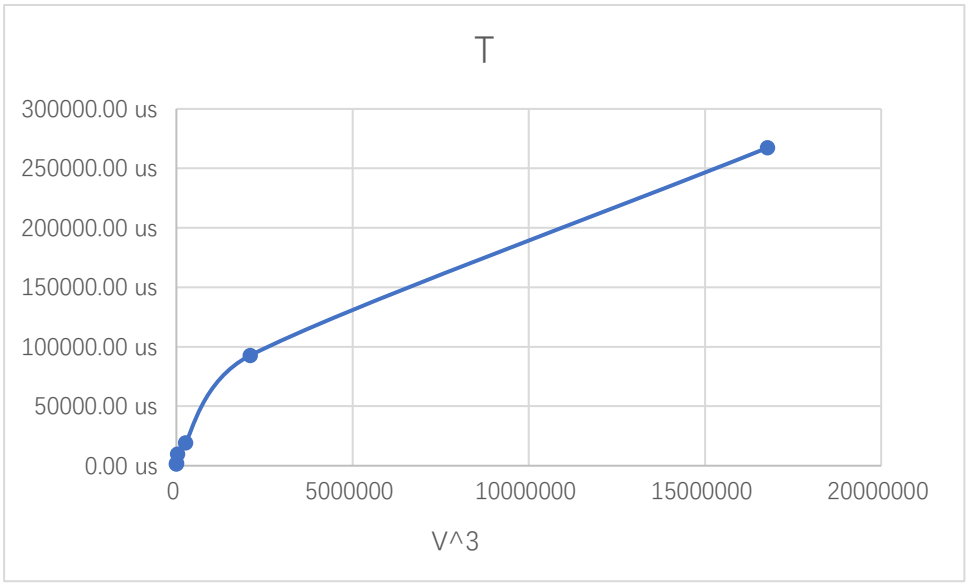
实验一中找到所有连通分量的时间（V+E = n+nlogn）

可以看出基本满足算法导论关于强连通分量算法Θ（V+E）的论断。



实验二中所有点对最短路径所消耗的时间(V=n)

可以看出在 n 较大时基本满足算法导论关于使用线性数组作为 Dijkstra 算法的优先队列时，JohnSon 算法的运行时间为O(V^3+VE)=O(V^3)的论断。



5. 实验心得

通过对强连通算法的实现，复习了表示图的邻接链表和邻接矩阵，复习了 DFS 算法，并对强连通算法有了较深的理解。通过对 JohnSon 算法的实现，复习了 Dijkstra 算法和优先队列，理解了 Bellman_Ford 算法。进一步学习了怎样测量程序运行时间以及图表的制作。