Project 4 Report

PB20111623 马子睿

实验内容与要求

实验内容

- 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：$\log_5 N$、 $\log_7 N$ 。不允许多重边，可以有环。

实验要求

- input.txt

  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。

- 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，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。

实验配置与环境

- 操作系统：Ubuntu22.10
- 处理器：11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- 时钟主频：2.80 GHz
- 编译器：gcc version 11.3.0

实验方法与步骤

实验方法

Johnson算法是基于Bellmanford算法和Dijkstra算法，以及对于边权值重新定义来计算所有点对之间的最短路径的算法。同时，由于Bellmanford算法无法处理负环的情况，因此我们还需要在运行算法之前切除掉所有的负环。

Bellmanford算法通过创建虚拟源节点计算对边进行松弛，使用每个点到虚节点的最短路径距离来重新赋予边权重，来避免负权边，使得Dijkstra失效。之后使用Dijkstra算法对每个顶点求解最短路径，最后恢复边权即可。

在本次实验中，我使用最小二叉堆来实现Dijkstra算法。由于C++的STL优先级队列没有减值操作，因此我手写实现了最小二叉堆。

实验步骤

构建图的数据结构

针对稀疏图的特点，我使用了邻接表来构建图：

```
class Graph{
    private:
    int  heapsize;
    void build_heap();
    void dfs_cut_minus_circle(bool visited[], int i, int w[]);
    public:
    vector<vnode>   vlist;
    vector<vnode>   vheap;
    vector<int>     map;
    Graph(int num);
    ~Graph();
    int     relax(int i, int j, int w);
    bool    BellmanFord(int s, bool do_cut);
    void    min_heapify(int i);
    void    decreace_key(int i, int key);
    int     extract_min();
    void    cut_minus_circle();
    void    Dijkstra(int s);
    void    Johnson();
    void    print_path(int s, int i, ofstream &fout);
};
```

其中，顶点的vnode数据结构为：

```
struct vnode{
    int vnum;
    int h;
    int dis;
    int p;
    vector<enode> *elist;
};
```

vnum为顶点编号，h为边权重构函数，dis 、p为供最短路径计算记录的最短路径距离和前驱节点编号。

其中，边的enode数据结构为：

```
struct enode{
    int vnum;
    int weight;
    int w;
    enode(){}
    enode(int _vnum, int _weight){
        vnum = _vnum;
        weight = _weight;
        w = weight;
    }
};
```

在这里，w是供Dijkstra算法使用的非负边权，而weight是真实的边权。

在Graph类中，我定义了所需的所有数据结构的方法，包括**最小二叉堆的建堆、取最小值、关键字减值，去除图中的负环、Dijkstra算法、Bellmanford算法和Johnson算法等**方法。

剪除负环边算法：

我使用深度优先搜索来剪除负环边：

```
void Graph::cut_minus_circle(){
    int n = vlist.size() - 1;
    bool *visited = new bool[n];
    int  *w = new int[n];

    for(int i = 0; i < vlist.size()-1; i++){
        memset(visited, false, n * sizeof(bool));
        memset(w, 0, n * sizeof(int));
        dfs_cut_minus_circle(visited, i, w);
    }
    delete[] visited;
    delete[] w;
}

void Graph::dfs_cut_minus_circle(bool visited[], int i, int w[]){
    if(visited[i]) return;
    visited[i] = true;
    for(auto j = vlist[i].elist->begin(); j != vlist[i].elist->end(); j++){
        if(visited[j->vnum]){
            if(w[i] - w[j->vnum] + j->weight < 0){
                // cout << "cut: " << i << " " << j->vnum << endl;
                vlist[i].elist->erase(j--);
            }
        }
        else {
            w[j->vnum] = w[i] + j->weight;
            dfs_cut_minus_circle(visited, j->vnum, w);
            w[j->vnum] = 0;
        }
    }
}
```

该算法利用了一个w数组，这个数组记录了深度遍历到当前的权重和，一旦找到了一个已经遍历过的节点，就证明已经找到了一个环，只需要利用找到环的节点加上最后这条边的权值，看看是否是负环。如果是负环，那么就剪除这条边，否则不予剪除。

这种方法的时间复杂度为$O(V \times (V + E))$。这一部分不计入算法时间中。

Bellmanford算法：

```
int Graph::relax(int i, int j, int w){
    int dis = (vlist[i].dis == INF) ? INF : vlist[i].dis + w;
    if(vlist[j].dis > dis){
        vlist[j].dis = dis;
        vlist[j].p = i;
    }
    return vlist[j].dis;
}
bool Graph::BellmanFord(int s, bool do_cut){
    int n = vlist.size();
    // initialize
    for(int i = 0; i < n; i++) {
        vlist[i].dis = INF;
        vlist[i].p = -1;
    }
    vlist[s].dis = 0;
    // relax every edge
    for(int k = 0; k < n - 1; k++){
        for(int i = 0; i < n; i++){
            for(auto j = vlist[i].elist->begin(); j != vlist[i].elist->end();
j++){
                relax(i, j->vnum, j->weight);
            }
        }
    }
    for(int i = 0; i < n; i++){
        for(auto j = vlist[i].elist->begin(); j != vlist[i].elist->end(); j++){
            int dis = (vlist[i].dis == INF) ? INF : vlist[i].dis + j->weight;
            if(vlist[j->vnum].dis > dis) {
                if(!do_cut) return false;
            }
        }
    }
    return true;
}
```

BellmanFord算法基本参照了书中的实现，这里做了一点扩展：最开始我希望使用Bellmanford算法来剪除所有负环，因此传入了do_cut变量，但显然这种方法是很难做到的。因此之后这个变量就变为开启负环检查的开关变量。

在这个算法之中，需要注意：**无穷加减任何有穷变量都是无穷**，这一点在之后的所有算法中都需要特别注意。

Dijkstra算法

我使用二叉堆对Dijkstra算法进行优化。这里遇到了一个朴素算法不会遇到的问题，请参照后文中**问题与解决**进行阅读。

Dijkstra借用二叉堆优化，可以非常简单的实现“提取最小值”和关键字减值：

```
void Graph::Dijkstra(int s){
    int n = vlist.size() - 1;
    for(int i = 0; i < n; i++) {
        vlist[i].dis = INF;
    }
    vlist[s].dis = 0;
    build_heap();
    while(heapsize > 0){
        int u = extract_min();
        for(auto i = vlist[u].elist->begin(); i != vlist[u].elist->end(); i++){
            decreace_key(map[i->vnum], relax(u, i->vnum, i->w));
        }
    }
    for(int i = 0; i < n; i++){
        pre[s][i] = vlist[i].p;
    }
    return;
}
```

Dijkstra算法基本借用书中的实现方法，需要注意的是，由于在Bellmanford算法中，我们加入了虚节点s，而在Dijkstra算法中这个节点已经不需要了。这里得益于我们的设计：**我们将这个虚节点直接push_back进图的节点列表——换句话说，它是编号等于节点列表长的节点，我们可以非常容易地将其忽略**。同时，经过我们的测量发现，C++自带的STL优先级队列也有一定的性能问题，因此这里我们选择手写实现。

Dijkstra算法中，使用了一个成员：map。这个变量请参见**问题与解决**中的解读。

Johnson算法

Johnson算法调用Bellmanford算法生成函数h，通过重构边权之后对每个节点调用Dijkstra算法构建最短路径矩阵。其中，边权重构公式为：

$$w[i,j] = weight[i,j] + h[i] - h[j]$$

```
void Graph::Johnson(){
    int n = vlist.size();
    for(int i = 0; i < n - 1; i++){
        vlist[n-1].elist->push_back(enode(i, 0));
    }
    if(!BellmanFord(n-1, false)){
        cout << "error! minus circle!" << endl;
        exit(-1);
    }
    for(int i = 0; i < n; i++){
        vlist[i].h = vlist[i].dis;
    }
    for(int i = 0; i < n; i++){
        for(auto j = vlist[i].elist->begin(); j != vlist[i].elist->end(); j++){
```

```
            if(j->weight == INF) j->w = INF;
            else j->w = j->weight + vlist[i].h - vlist[j->vnum].h;
        }
    }
    for(int i = 0; i < n - 1; i++){
        Dijkstra(i);
        for(int j = 0; j < n - 1; j++){
            if(vlist[j].dis == INF) min_dis[i][j] = INF;
            else min_dis[i][j] = vlist[j].dis + vlist[j].h - vlist[i].h;
        }
    }
    vlist[n-1].elist->clear();
}
```

首先调用Bellmanford算法来计算虚节点到所有节点的最短路径，用来构造新边权。构造边权之后对每个节点调用Dijkstra算法构建最短路径矩阵即可。

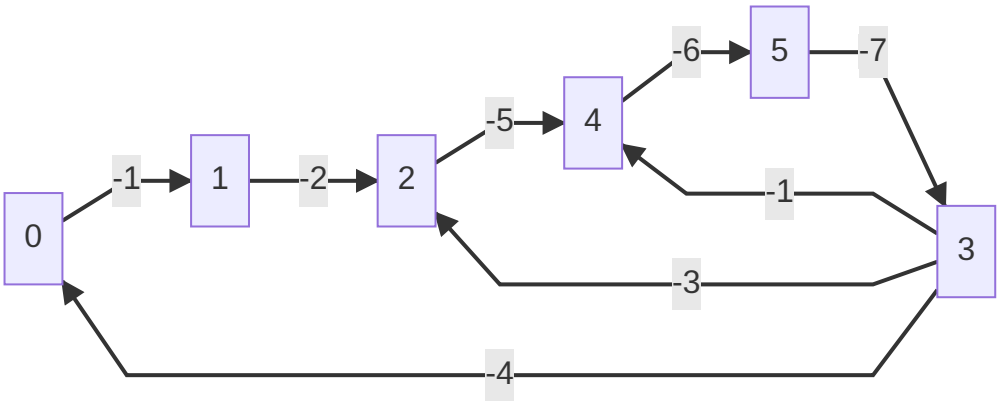
实验结果与分析

正确性验证

我们使用一个简单的测试用例来验证基本的正确性：

```
0 1 -1
1 2 -2
3 2 -3
3 0 -4
2 4 -5
4 5 -6
5 3 -7
3 4 -1
```

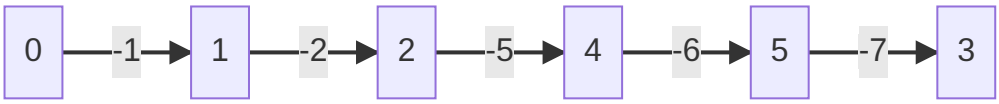
其图如下：



程序剪除的边为：

```
cut: 3 2
cut: 3 0
cut: 3 4
```

因此剪除后的图为



最终结果为：

```
0   0  : 0  0
0   1  : 0, 1  -1
0   2  : 0, 1, 2  -3
0   3  : 0, 1, 2, 4, 5, 3  -21
0   4  : 0, 1, 2, 4  -8
0   5  : 0, 1, 2, 4, 5  -14
1   0  : unreachable
1   1  : 1  0
1   2  : 1, 2  -2
1   3  : 1, 2, 4, 5, 3  -20
1   4  : 1, 2, 4  -7
1   5  : 1, 2, 4, 5  -13
2   0  : unreachable
2   1  : unreachable
2   2  : 2  0
2   3  : 2, 4, 5, 3  -18
2   4  : 2, 4  -5
2   5  : 2, 4, 5  -11
3   0  : unreachable
3   1  : unreachable
3   2  : unreachable
3   3  : 3  0
3   4  : unreachable
3   5  : unreachable
4   0  : unreachable
4   1  : unreachable
4   2  : unreachable
4   3  : 4, 5, 3  -13
4   4  : 4  0
4   5  : 4, 5  -6
5   0  : unreachable
5   1  : unreachable
5   2  : unreachable
5   3  : 5, 3  -7
5   4  : unreachable
5   5  : 5  0
```

可以验证，这样的结果是正确的。

为了进一步验证正确性，我对某次27节点的图进行了验证，验证过程比较复杂，这里就不再赘述了，实验要求的所有输出已经存放在output文件夹下。

时间复杂度分析

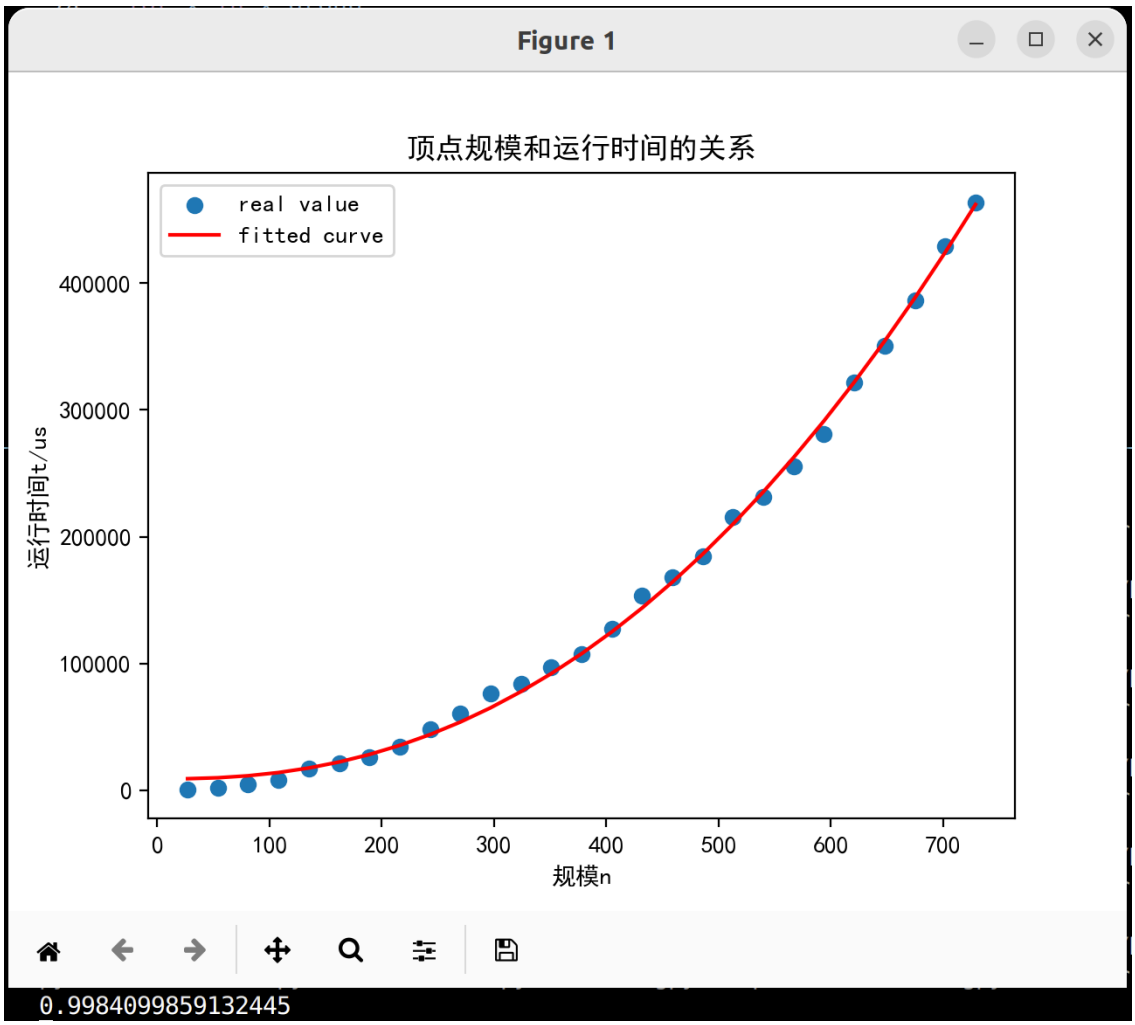
我们使用python脚本对曲线进行拟合，并分析其拟合优度，具体代码请参见src/fitting.py

由于只有四组数据难以拟合曲线，因此我选择了对规模为27的倍数都构建了用例，具体请看time_test文件夹

- 对顶点度数为$log_5 n$的情况，测得数据如下：

规模	运行时间1	运行时间2	运行时间3	运行时间4	平均运行时间
27	680	287	1055	390	603
54	1964	895	3546	2225	2157.5
81	4604	2920	8381	3493	4849.5
108	7657	3391	14535	6187	7942.5
135	18695	7467	32093	11114	17342.25
162	25454	10502	33682	15547	21296.25
189	35315	14600	33021	20958	25973.5
216	44360	19350	44131	28403	34061
243	56559	24567	54275	55443	47711
270	66718	30258	79619	65290	60471.25
297	78001	36414	115349	76673	76609.25
324	90771	42498	104755	97012	83759
351	106125	52677	111543	118746	97272.75
378	115222	61518	129640	122973	107338.25
405	139592	70033	150025	150468	127529.5
432	165179	76507	165620	207277	153645.75
459	182617	96011	176854	215199	167670.25
486	201006	133092	196859	207760	184679.25
513	256135	153982	221526	229096	215184.75
540	247918	170523	262658	245562	231665.25
567	277727	186063	279312	279609	255677.75
594	293594	241782	292400	296214	280997.5
621	321680	303564	309043	352246	321633.25
648	414074	306437	309801	373220	350883
675	453817	326288	320579	445251	386483.75
702	485329	343956	322161	486572	409504.5
729	506003	400268	425198	523780	463812.25

使用脚本对$y = A \times n^2 log_5 n log_2 n + C$进行曲线拟合可得：

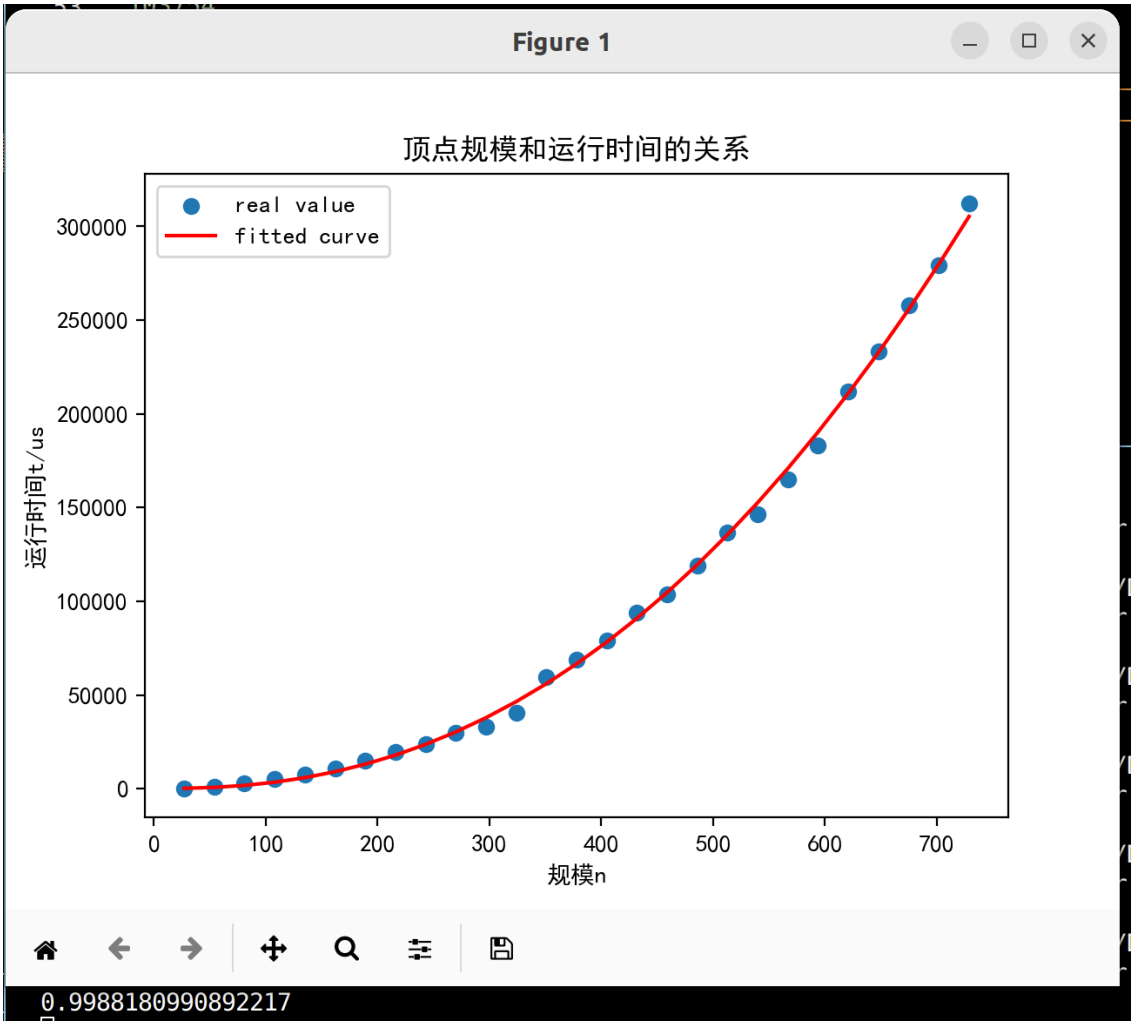


最下方为拟合优度，可以看出，算法基本符合$O(VElog_2 V)$的时间复杂度

- 对顶点度数为$log_7 n$的情况，测得数据如下：

规模	运行时间1	运行时间2	运行时间3	运行时间4	平均运行时间
27	206	431	219	252	277
54	923	2008	941	926	1199. 5
81	2121	4419	2027	2090	2664. 25
108	3702	8850	3677	3775	5001
135	5754	12939	6375	5631	7674. 75
162	7931	18237	8105	7993	10566. 5
189	10659	25776	11015	11394	14711
216	15043	32376	14819	15209	19361. 75
243	18223	37546	19078	20030	23719. 25
270	23922	48949	22944	23055	29717. 5
297	27558	51577	26607	26607	33087. 25
324	35344	63349	31924	32007	40656
351	49440	91180	49767	48356	59685. 75
378	60002	104530	55329	56154	69003. 75
405	68078	118605	65019	64828	79132. 5
432	80775	138060	80441	77280	94139
459	85894	157641	85942	85539	103754
486	96600	173041	110358	96230	119057. 25
513	110145	194195	132653	110165	136789. 5
540	116399	216756	139530	113476	146540. 25
567	133403	237562	156604	132823	165098
594	145871	246688	170473	170329	183340. 25
621	169776	284343	194972	199120	212052. 75
648	176002	313235	207782	235643	233165. 5
675	195427	330641	226052	279598	257929. 5
702	207888	335739	232009	341673	279327. 25
729	237247	375503	248543	388328	312405. 25

使用脚本对$y = A \times n^2 log_7 n log_2 n + C$进行曲线拟合可得：



问题与解决

- 最小二叉堆的构建

在图结构中，我们使用顶点编号来索引顶点的边列表。图结构中，若顶点编号为i，则其一定在顶点列表的第i个位置，这就为O(1)索引到顶点提供了可能。但是，一旦创建最小二叉堆，那么顶点编号就被打乱了。如果还想直接索引顶点，那么必须使用一个映射来记录i号顶点在列表的哪个位置。需要维护map的操作为：

- min_heapify：在交换顶点位置时，需要维护map：

```
void Graph::min_heapify(int i){
    int l = i * 2 + 1;
    int r = i * 2 + 2;
    int min;
    if(l < heapsize && vheap[l].dis < vheap[i].dis) min = l;
    else min = i;
    if(r < heapsize && vheap[r].dis < vheap[i].dis) min = r;
    while(min != i){
        swap(vheap[i], vheap[min]);
        map[vheap[i].vnum] = i;
        map[vheap[min].vnum] = min;
        i = min;
        l = i * 2 + 1;
        r = i * 2 + 2;
        if(l < heapsize && vheap[l].dis < vheap[i].dis) min = l;
        else min = i;
        if(r < heapsize && vheap[r].dis < vheap[i].dis) min = r;
    }
}
```

- decrease_key：在交换顶点位置时，需要维护map：

```
void Graph::decreace_key(int i, int key){
    if(vheap[i].dis < key)
        while(1);
    vheap[i].dis = key;
    while(i > 0 && vheap[(i - 1) / 2].dis > vheap[i].dis){
        swap(vheap[(i - 1) / 2], vheap[i]);
        map[vheap[(i - 1) / 2].vnum] = (i - 1) /  2;
        map[vheap[i].vnum] = i;
        i = (i - 1) / 2;
    }
}
```

- extract_min：在交换顶点位置时，需要维护map：

```
int Graph::extract_min(){
    int min = vheap[0].vnum;
    swap(vheap[0], vheap[heapsize-1]);
    map[vheap[0].vnum] = 0;
    map[vheap[heapsize-1].vnum] = heapsize - 1;

    heapsize--;
    return min;
}
```

在使用Dijkstra算法时，需要通过map来锁定需要减值的关键字在顶点列表中的位置：

```
while(heapsize > 0){
    int u = extract_min();
    for(auto i = vlist[u].elist->begin(); i != vlist[u].elist->end(); i++){
        decreace_key(map[i->vnum], relax(u, i->vnum, i->w));
    }
}
```

由此，便解决了最小堆化打乱节点顺序的问题。

实验总结

通过本次实验，我有了如下收获：

- 对图算法有了更深刻的理解。此次实验集各大最短路径算法于大成，让我对图算法有了统领全局的理解。
- 熟悉了python分析数据的操作，并感受到了python拟合曲线的便捷。