实验4 图论算法

实验设备和环境

macOS Mojave 10.14.6 MacBook Pro (Retina, 13-inch, Early 2015) 处理器 2.9 GHz Intel Core i5

IDE Clion

实验4.1：Kruskal算法

实验要求

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4.1 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$1 + \text{rand}()\%\lfloor N/2 \rfloor$。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。

实验过程

Kruskal 算法的伪代码如下图

```
MST-KRUSKAL(G,w)
1  A=∅
2  for each vertex v∈G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge(u,v)∈G.E, taken in nondecreasing order by weight
6      if FIND-SET(v)≠FIND-SET(v)
7          A=A∪{(u,v)}
8          UNION(u,v)
9  return A
```

用到 MAKE-SET 和 FIND-SET 等操作，所以需要引入并查集的数据结构；在第 4 行 "sort edges" 的时候，涉及到边排序，其实不需要引入邻接矩阵或邻接表这样的图结构。具体实现如下：

```c
struct EdgeType{
    int i;
    int j;
    int w;
};

struct EdgeGraph{
    int vertexNum;
    int edgeNum;
    struct EdgeType edge[MaxEdge];
};
```

边结构记录两个端点和权重，图结构记录结点数量、边数量和所有边。

对不相交集合的处理如下（模仿课本伪代码的实现）：

```c
int father[MAX];
int Rank[MAX];

void MAKE_SET(int x){
    father[x] = x;
    Rank[x] = 0;
}

int FIND_SET(int x){
    if(x != father[x])
        father[x] = FIND_SET(father[x]);
    return father[x];
}

void UNION(int x, int y){
    if(x == y)
        return;
    if(Rank[x] > Rank[y])
        father[y] = x;
```

```
    else{
        if(Rank[x] == Rank[y])
            Rank[y]++;
        father[x] = y;
    }
}
```

本实验要求的输入构造函数如下：

```
bool find_edge(int i, int j){
    for(int k = 0; k < G.edgeNum; k++){
        if((G.edge[k].i == i && G.edge[k].j == j) ||\
        (G.edge[k].i == j && G.edge[k].j == i))
            return true;
    }
    return false;
}

void rand_input(FILE *fp, int N){
    // i, j, w_ij~(1, 20)
    // 1+rand()%(N/2)
    // num for every i
    srand((unsigned)time(NULL));
    int num, w, i, j;
    char buf[BUF_LEN];
    int *p = (int *)malloc(sizeof(int)*N);
    int max_num = (int)(N/2);
    int place = 0;
    memset(p, 0, sizeof(int)*N);
    for(i = 0; i < N; i++){
        num = 1+rand()% max_num;
        if(p[i]+num > max_num)
            num = max_num-p[i];
        for(int k = 0; k < num; k++){
            do{
                j = rand()%N;
            }while(j == i);
            if(find_edge(i, j)) {
                //printf("find one\n");
                continue;
            }
            if(p[j] >= max_num)
                continue;
            w = 1+rand()%20;
            p[j] = p[j]+1;
            p[i] = p[i]+1;
            memset(buf, '\0', BUF_LEN * sizeof(char));
            sprintf(buf, "%d %d %d\n", i, j, w);
```

```
            //printf("%d %d %d\n", i, j, w);
            fprintf(fp, "%s", buf);
            G.edge[place].i = i;
            G.edge[place].j = j;
            G.edge[place].w = w;
            place++;
            G.edgeNum++;
        }
    }
    free(p);
}
```

核心思路是随机确定某点需要添加出边数量 num，如果加上 num 会超出限定 $\lfloor N \rfloor$，就修改 num 使其不超出限定。之后，生成对应数量的边，如果边是重边（利用 find_edge）直接去掉（有重边说明该结点边数量 >= 1，已经满足要求）。本函数同时生成对应的 input 文件和前文提及的图结构。

最核心的当然是 Kruskal 算法：

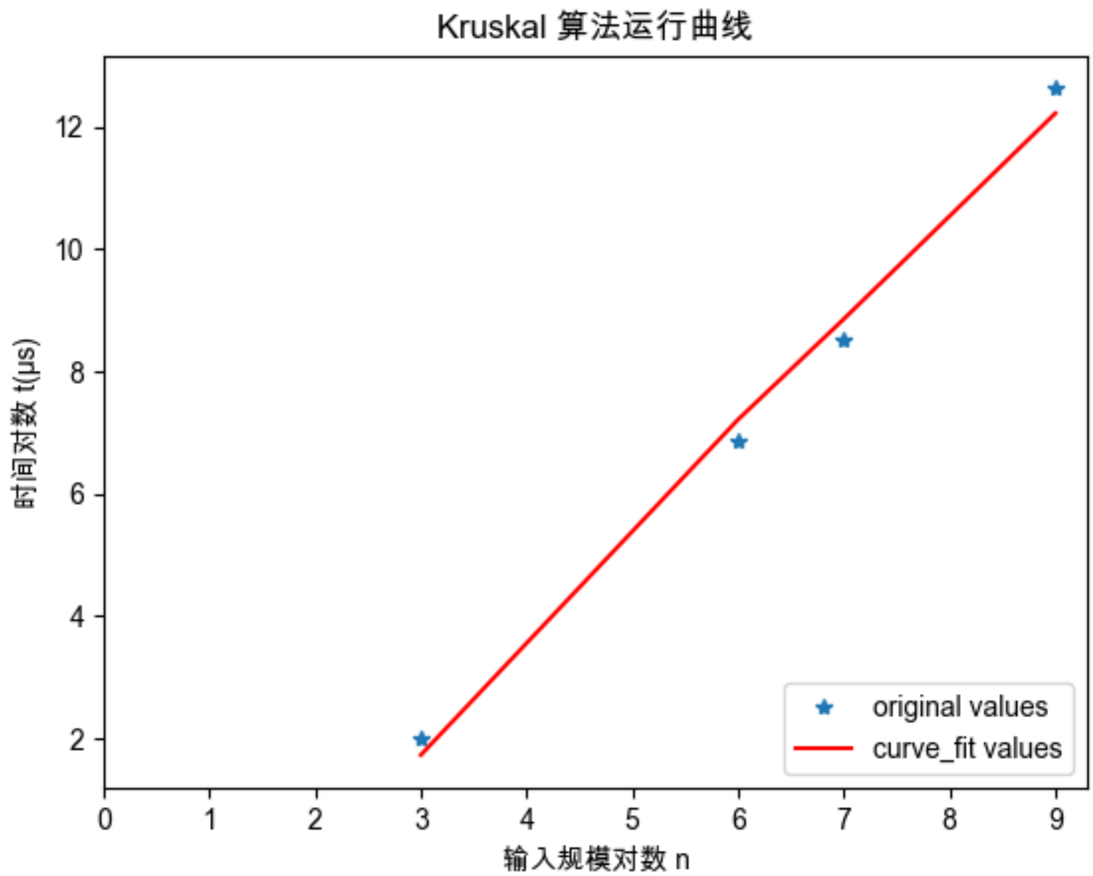
```
bool comp(EdgeType x, EdgeType y){
    return x.w < y.w;
}

EdgeType * MST_KRUSKAL(){
    EdgeType *A = (EdgeType *)malloc(sizeof(EdgeType)*(G.vertexNum-1));
    int place = 0;

    for(int i = 0; i < G.vertexNum; i++)
        MAKE_SET(i);

    sort(G.edge, G.edge+G.edgeNum, comp);
    int from, to;
    for(int i = 0; i < G.edgeNum; i++){
        from = FIND_SET(G.edge[i].i);
        to = FIND_SET(G.edge[i].j);
        if(from != to){
            A[place] = G.edge[i];
            place++;
            UNION(from, to);
        }
    }
    return A;
}
```

这里有用到 cpp algorithm 库中的 sort() 函数，复杂度并不比用最小堆更高，可以使用。

其他的文件 I/O 和动态分配空间释放相关都略过。

实验结果

比较实际复杂度和理论复杂度是否相同，给出分析。



可以看到实际与理论拟合效果很好，可以认为相同。

实验4.2：Johnson算法

实验要求

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 ， 同一顶点数目对应两种弧的数目: $log_5 N$和 $log_7 N$(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4.2 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。

实验4.2 Johnson算法

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

实验过程

```
BELLMAN-FORD(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
for i=1 to |G.V|-1
    for each edge(u,v)∈G.E
        RELAX(u,v,w)
for each edge(u,v)∈G.E
    if v.d>u.d+w(u.v)
        return FALSE
return TRUE
```

```
DIJKSTRA.(G,w,s)
INITIALIZE-SINGLE-SOURCE(G,s)
S=∅
Q=G.V
while Q≠∅
    u=EXTRACT-MIN(Q)
    S=S∪{u}
    for each vertex v∈G.Adj[u]
        RELAX(u,v,w)
```

```
JOHNSON(G,w)
compute G′,where G′.V=G.V∪{s},
        G′.E=G.E∪{(s,v):v∈G.V},and
        w(s,v)=0 for all v∈G.V
if BELLMAN-FORD(G′,w,s)==FALSE
    print"the input graph contains a negative-weight cycle"
else for each vertex v∈G′.V
        set h(v)to the value of δ(s,v)
            computed by the Bellman-Ford algorithm
    for each edge(u,v)∈G′.E
        ŵ(u,v)=w(u,v)+h(u)-h(v)
    let D=(d_uv)be a new n×n matrix
    for each vertex u∈G.V
        run DIJKSTRA(G,ŵ,u)to compute δ̂(u,v)for all v∈G.V
        for each vertex v∈G.V
            d_uv=δ̂(u,v)+h(v)-h(u)
    return D
```

数据结构设计如下:

```
typedef struct ArcNode{
    int from, to, w;
    struct ArcNode *next;
}ArcNode;

typedef struct VertexNode{
    int id;
    int key; // for BellmanFord's d
    VertexNode *p; // for BellmanFord's parent
    ArcNode *firstArc;
}VertexNode;

typedef struct Graph{
    int VertexNum;
    VertexNode *Vertex;
}Graph;
```

ArcNode 是有向边的边结构，next 用于表达类似邻接表的链表结构；VertexNode 是结点结构，id 用于最小优先队列的顺序记录，key 是结点的值，p 记录 parent 父节点，firstArc 是结点的第一条出边，并可以沿路访问到所有出边。Graph 是图结构。

对图初始化:

```
void InitGraph(int VertexNum){
    G.VertexNum = VertexNum;
    G.Vertex = (VertexNode *)malloc(sizeof(VertexNode)*G.VertexNum);
    for(int i = 0; i < G.VertexNum; i++)
        G.Vertex[i].firstArc = NULL;
}
```

构造随机输入利用了 cpp shuffle 函数来对数组重排（目的结点随机且不同）：

```
void rand_input(FILE *fp, int N, int type){
    char buf[BUF_LEN];

    int edgeNum;
    if(type == 0) {
        edgeNum = log(N)/log(5);
    }
    else if(type == 1)
        edgeNum = log(N)/log(7);
    else{
        printf("error!\n");
        exit(1);
    }

```

```
    int *p = (int *)malloc(sizeof(int)*N);
    memset(p, 0, sizeof(int)*N);
    int w, from, to;
    std::vector<int> v;
    ArcNode *ptr, *ptr_save;

    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    for(int i = 0; i < N; i++)
        v.push_back(i);
    for(int i = 0; i < N; i++){
        std::shuffle(v.begin(), v.end(), std::default_random_engine(seed));
        G.Vertex[i].firstArc = (ArcNode *)malloc(sizeof(ArcNode));
        ptr = G.Vertex[i].firstArc;
        ptr->next = NULL;
        while(p[i] < edgeNum){
            //w = -10+rand()%61;
            w = rand()%51;
            from = i;
            to = v[p[i]];
            p[i] = p[i]+1;
            memset(buf, '\0', BUF_LEN * sizeof(char));
            sprintf(buf, "%d %d %d\n", from, to, w);
            fprintf(fp, "%s", buf);
            ptr->from = from;
            ptr->to = to;
            ptr->w = w;
            ptr->next = (ArcNode *)malloc(sizeof(ArcNode));
            ptr_save = ptr;
            ptr = ptr->next;
        }
        free(ptr_save->next);
        ptr_save->next = NULL;
    }
}
```

把源结点 s 放在图结点数组的最后一个位置，Johnson 算法中对源节点有处理如下：

```
G.Vertex[N].firstArc = (ArcNode *)malloc(sizeof(ArcNode));
    ptr = G.Vertex[N].firstArc;
    for(int k = 0; k < N-1; k++){
        ptr->from = N;
        ptr->to = k;
        ptr->w = 0;
        ptr->next = (ArcNode *)malloc(sizeof(ArcNode));
        ptr = ptr->next;
    }
    ptr->from = N;
    ptr->to = N-1;
    ptr->w = 0;
    ptr->next = NULL;
```

BELLMAN-FORD 算法的实现和伪代码几乎一致，只需根据数据结构调整具体代码即可。

```
void RELAX(ArcNode *arc){
    if(G.Vertex[arc->to].key > G.Vertex[arc->from].key+arc->w){
        G.Vertex[arc->to].key = G.Vertex[arc->from].key+arc->w;
        G.Vertex[arc->to].p = &G.Vertex[arc->from];
    }
}

bool BELLMAN_FORD(){
    INITIALIZE_SINGLE_SOURCE(G.VertexNum-1);
    for(int i = 0; i < G.VertexNum-1; i++){
        for(int j = 0; j < G.VertexNum; j++)
            for(ArcNode *arc = G.Vertex[j].firstArc; arc != NULL; arc = arc->next)
                RELAX(arc);
    }
    for(int j = 0; j < G.VertexNum; j++)
        for(ArcNode *arc = G.Vertex[j].firstArc; arc != NULL; arc = arc->next){
            if(G.Vertex[arc->to].key > G.Vertex[arc->from].key+arc->w)
                return false;
        }
    return true;
}
```

因为输入没有负边，所以 BELLMAN_FORD 算法的结果肯定是 true（预测与实际一致）：

```
int main(){
    ...
    if(!BELLMAN_FORD()){
            printf("false!\n");
        }
    int *h = (int *)malloc(sizeof(int)*G.VertexNum);
```

```
    for(int i = 0; i < G.VertexNum; i++)
        h[i] = G.Vertex[i].key;
    for(int i = 0; i < G.VertexNum; i++){
        ArcNode *arc = G.Vertex[i].firstArc;
        for(; arc != NULL; arc = arc->next)
            arc->w = arc->w+h[arc->from]-h[arc->to];
    }
    ...
}
```

根据课本关于优先队列的伪代码构造最小堆情况如下：

```
int PARENT(int i){
    return (i-1)/2;
}
int LEFT(int i){
    return 2*i+1;
}
int RIGHT(int i){
    return 2*i+2;
}

void MIN_HEAPIFY(VertexNode** A, int i, int heapsize){
    int l, r, largest;
    VertexNode *t;
    l = LEFT(i);
    r = RIGHT(i);
    if(l < heapsize && A[l]->key < A[i]->key)
        largest = l;
    else
        largest = i;
    if(r < heapsize && A[r]->key < A[largest]->key)
        largest = r;
    if(largest != i){
        t = A[i];
        A[i] = A[largest];
        A[i]->id = i;
        A[largest] = t;
        A[largest]->id = largest;
        MIN_HEAPIFY(A, largest, heapsize);
    }
}

VertexNode* HEAP_EXTRACT_MIN(VertexNode** A, int heapsize){
    if(heapsize < 1){
        printf("heap underflow\n");
        return NULL;
    }
```

```
    VertexNode *min = A[0];
    A[0] = A[heapsize-1];
    heapsize--;
    MIN_HEAPIFY(A, 0, heapsize);
    return min;
}

void HEAP_DECREASE_KEY(VertexNode** A, int i, int key){
    VertexNode *t;
    if(key > A[i]->key){
        printf("new key is larger than current key\n");
        return;
    }
    A[i]->key = key;
    while(i > 0 & A[PARENT(i)]->key > A[i]->key){
        t = A[i];
        A[i] = A[PARENT(i)];
        A[i]->id = i;
        A[PARENT(i)] = t;
        A[PARENT(i)]->id = PARENT(i);
        i = PARENT(i);
    }
}

void BUILD_MIN_HEAP(VertexNode** A, int heapsize){
    for(int i = heapsize/2; i >= 0; i--)
        MIN_HEAPIFY(A, i, heapsize);
}

void Dijkstra(int index){
    INITIALIZE_SINGLE_SOURCE(index);
    //printGraph();
    VertexNode **queue = (VertexNode**)malloc(sizeof(VertexNode*)*
(G.VertexNum-1));
    for(int i = 0; i < G.VertexNum-1; i++)
        queue[i] = &G.Vertex[i];

    BUILD_MIN_HEAP(queue, G.VertexNum-1);
    for(int i = 0; i < G.VertexNum-1; i++){
        VertexNode *node = HEAP_EXTRACT_MIN(queue, G.VertexNum-1);

        for(ArcNode *arc = node->firstArc; arc != NULL; arc = arc->next)
            if(G.Vertex[arc->to].key > G.Vertex[arc->from].key+arc->w){
                //G.Vertex[arc->to].key = G.Vertex[arc->from].key+arc->w;
                HEAP_DECREASE_KEY(queue, G.Vertex[arc->to].id,
G.Vertex[arc->from].key+arc->w);
                G.Vertex[arc->to].p = &G.Vertex[arc->from];
            }
    }
```

```
}
```

其中 id 记录的是在堆中的位置，而非结点的编号，所以在 printPath 中结点编号利用 firstarc->from 得到：

```
bool printPath(VertexNode *src, VertexNode *des, FILE *fp){
    char buf[BUF_LEN];
    bool flag;
    if(src == des){
        //printf("%d,", src->id);
        //fflush(stdout);
        memset(buf, '\0', BUF_LEN * sizeof(char));
        sprintf(buf, "%d ", src->firstArc->from);
        fprintf(fp, "%s", buf);
    }
    else if(des->p == NULL){
        //printf("no path");
        //fflush(stdout);
        memset(buf, '\0', BUF_LEN * sizeof(char));
        sprintf(buf, "no path");
        fprintf(fp, "%s", buf);
        return false;

    }
    else{
        flag = printPath(src, des->p, fp);
        if(!flag)
            return flag;
        //printf("%d ", des->id);
        //fflush(stdout);
        memset(buf, '\0', BUF_LEN * sizeof(char));
        sprintf(buf, "%d ", des->firstArc->from);
        fprintf(fp, "%s", buf);
    }
    return true;
}
```

值得一提的是，单源最短路径需要及时 printPath。一个我 de 了两小时的问题就来源于此（落泪

其他的文件 I/O 、动态分配空间释放（例如 freeGraph）、调试函数（例如 printGraph）相关都略过。

实验结果

比较实际复杂度和理论复杂度是否相同，给出分析。

值得一提的是，我在对 Dijstra 算法做处理的时候，试过一个复杂度不是 O(VlgV) 而是 O(V^2) 的实现（保留在 main.cpp 注释中）

```
bool cmp(VertexNode *node1, VertexNode *node2){
```

```
    return node1->key > node2->key;
}

void Dijkstra(int index){
    INITIALIZE_SINGLE_SOURCE(index);
    //printGraph();
    std::vector<VertexNode *> queue;
    for(int i = 0; i < G.VertexNum-1; i++)
        queue.push_back(&G.Vertex[i]);
    std::make_heap(queue.begin(), queue.end(), cmp);
    while(!queue.empty()){
        VertexNode *node = queue.front();
        std::pop_heap(queue.begin(), queue.end(), cmp);
        queue.pop_back();
        for(ArcNode *arc = node->firstArc; arc != NULL; arc = arc->next)
            RELAX(arc);
        std::make_heap(queue.begin(), queue.end(), cmp);
    }
}
```

time.txt 数据对比如下：

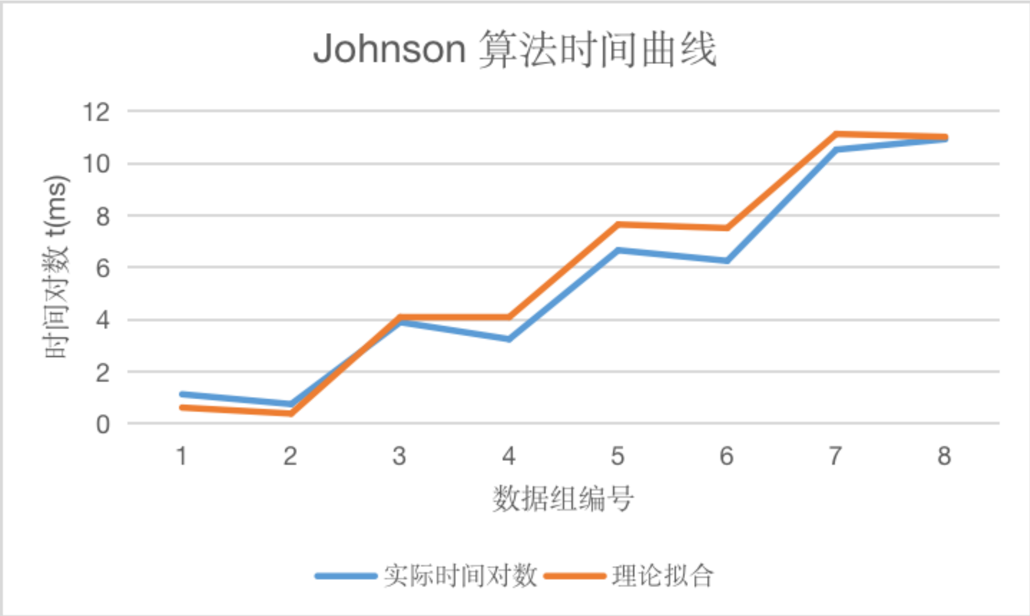
before->

```
0.004016
0.002409
0.045519
0.049515
0.777806
0.743053
12.811507
12.183733
```

after->

```
0.002387
0.002504
0.014566
0.019685
0.274708
0.180602
1.417714
1.925438
```

利用 Johnson 算法理论复杂度为 O($VElgV$) 作拟合，观察到实际与理论拟合效果很好，可以认为相同。


Johnson 算法时间曲线
时间对数 t(ms)
0
2
4
6
8
10
12
1
2
3
4
5
6
7
8
数据组编号
实际时间对数
理论拟合