实验四 图论算法

学号：PB18000268　姓名：曾勇程

实验设备和环境

- **硬件：** Lenovo LEGION Y7000 处理器 Intel Core i7 CPU @ 2.20GHz 2.21GHz
- **软件：** Windows 10 64位，Dev-C++ 5.11

实验4.1 Kruskal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4.1 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。

实验方法和步骤

- 首先随机生成图(无向图，每个顶点[1, N/2]条边):

```
int N;  // 结点数
vector<pair<int, int> >E;    // 边集合
```

```cpp
vector<int> weights;        // 边对应的权重

void Rand_Input(int n, const char *path){
    int i, j, k, x;
    int node1, node2, edge_number, edge;
    int weight;
    bool flag = true;       // 标志是不是所有顶点的关联边数是否还可以增加
    int count[n];   // 每个结点关联边数
    FILE* fp;
    pair<int, int> rand_edge;    // 随机生成的边
    pair<int, int> sym_edge;    // 随机生成的边的对称边,判断生成的边是否为重边
    for(i = 0; i < n; i++)      // 初始化
        count[i] = 0;

    srand((unsigned)time(NULL));    // 用系统定时/计数器的值作为随机种子，产生比较好的随机数
    for(i=0; i<n; i++){
        rand_edge.first = i;
        sym_edge.second = i;
        edge_number = rand() % (n/2) + 1;   // 对每一顶点随机生成的边数
        edge = count[i];        // 目前已有边数
        while(edge < edge_number && flag){     // 边数是否达标，所有顶点的边数是否还可以增加
            rand_edge.second = rand() % n;          // 选择另外一个顶点
            sym_edge.first = rand_edge.second;
            weight = rand() % 20 + 1;     // 边的权重为 (1, 20) 包括 1 和 20
            if(find(E.begin(), E.end(), rand_edge) == E.end() &&
                find(E.begin(), E.end(), sym_edge) == E.end() &&
                count[rand_edge.second] < n/2)
            {  // 无重边且另一个顶点的相关边少于 N/2 条
                E.push_back(rand_edge);          // 加入这条边
                weights.push_back(weight);
                edge += 1;
                count[i] += 1;
                if(rand_edge.second != i)   // 指向自身的环边只算一条
                    count[rand_edge.second] += 1;
            }
            else{
                continue;
            }
            for(j = 0; j < n; j++){     // 判断所有顶点的边数是否已达上界
                if(count[j] < n/2)
                    break;
            }
            if(j >= n)
                flag = false;
        }
    }
    if((fp=fopen(path,"w"))==NULL){
        printf("Fail to open file!\n");
        exit(0);
    }
    for(k = 0; k < E.size(); k++ ){
        fprintf(fp,"%d\t%d\t%d\n",E[k].first, E[k].second, weights[k]);
        printf("%d\n",k+1);
    }
    E.clear();  // 清空边
    weights.clear();    // 清空权重
```

```
    printf("OK!!!\n");
    fclose(fp);
    return;
}

int main(){
    const char *path;
    cout << "Please enter the number of vertices(N:0, 8, 64, 128, 512):" << endl;
    cin >> N;
    while(N){
        switch (N){
        case 8:
            path = "../input/input1.txt";
            break;
        case 64:
            path = "../input/input2.txt";
            break;
        case 128:
            path = "../input/input3.txt";
            break;
        case 512:
            path = "../input/input4.txt";
            break;
        default:
            cout << "Input Error!" << endl;
            cout << "Please enter the number of vertices(N:0, 8, 64, 128, 512):" << endl;
            cin >> N;
            continue;
        }
        Rand_Input(N, path);
        cout << "Please enter the number of vertices(N:0, 8, 64, 128, 512):" << endl;
        cin >> N;
    }
    return 0;
}
```

根据输入生成input1.txt, input2.txt,.....,input4.txt （讲解可见注释）。

- Kruskal算法的数据结构：

```
int N;  // 结点数

typedef struct node{
    int key;    // 关键字
    int rank;   // 属于哪个集合
    node* p;    // 前驱
}Node;

typedef struct graph{
    Node **V;   // 顶点集合
    vector<pair<pair<int, int>, int> > E;    // 边集合
}Graph, *GraphPtr;
```

- 不相交集合的相关操作：

```
Node *MAKE_SET(int key){
    Node* x = (Node *)malloc(sizeof(Node));
    x->key = key;
    x->p = x;
    x->rank = 0;
    return x;
}

Node *FIND_SET(Node* x){
    if (x->p!=x)
        x->p = FIND_SET(x->p);
    return x->p;
}

void LINK(Node* x, Node* y)
{
    if(x->rank > y->rank)
        y->p = x;
    else{
        x->p = y;
        if(x->rank==y->rank){
            y->rank += 1;
        }
    }
}


void UNION(Node* x, Node* y){
    LINK (FIND_SET(x), FIND_SET(y));
}
```

模仿课本21章伪代码完成。

- Kruskal算法：

```
bool compare(pair<pair<int, int>, int> x, pair<pair<int, int>, int> y){
// 排序方法
    return(x.second < y.second);
}

vector<pair<pair<int, int>, int> > MST_KRUSKAL(GraphPtr G){
    vector<pair<pair<int, int>, int> > A; // A
    for (int i = 0; i < N; i++){
        G->V[i] = MAKE_SET(i);      // 创建初始不相交集合
    }
    sort(G->E.begin(), G->E.end(), compare);    // 复杂度为O(nlgn)
    for (int i = 0; i < G->E.size(); i++){
        if (FIND_SET(G->V[G->E[i].first.first]) != FIND_SET(G->V[G-
>E[i].first.second])){   // 非同一个集合
            A.push_back(G->E[i]);
            UNION(G->V[G->E[i].first.first], G->V[G->E[i].first.second]);
```

```
        }
    }
    return A;
}
```

按照课本的伪代码实现。其中，排序直接调用了库函数`sort()`，其时间复杂度为O(nlgn)，符合课本。

- 构建边集：

```
void Get_Edges(const char *path, GraphPtr G){
    int i;
    FILE *fp;
    pair<pair<int, int>, int>edge;    // 边
    if((fp=fopen(path,"r"))==NULL){
        printf("Fail to open file!\n");
        exit(0);
    }
    while(fscanf(fp, "%d\t%d\t%d", &edge.first.first, &edge.first.second, &edge.second) != EOF){
        G->E.push_back(edge);
    }
    G->V = (Node **)malloc(sizeof(Node *)*N);   // 为结点集合分配空间
    fclose(fp);
}
```

- `main`函数：

```
int main(){
    int i, tree_weight;
    const char *path, *outpath;
    Graph G;    // 图
    FILE *fp_time, *fp_result;
    vector<pair<pair<int, int>, int> > A;   // MST的边集

    double run_time, start, finish;
    _LARGE_INTEGER time_start;  //开始时间
    _LARGE_INTEGER time_over;   //结束时间
    double dqFreq;      //计时器频率
    LARGE_INTEGER f;    //计时器频率

    cout << "Please enter the number of vertices(N:0, 8, 64, 128, 512):" << endl;
    cin >> N;

    if((fp_time = fopen("../output/time.txt","w"))==NULL){
        printf("Fail to open time.txt!\n");
        exit(0);
    }

    while(N){
        switch (N){
        case 8:
            path = "../input/input1.txt";
```

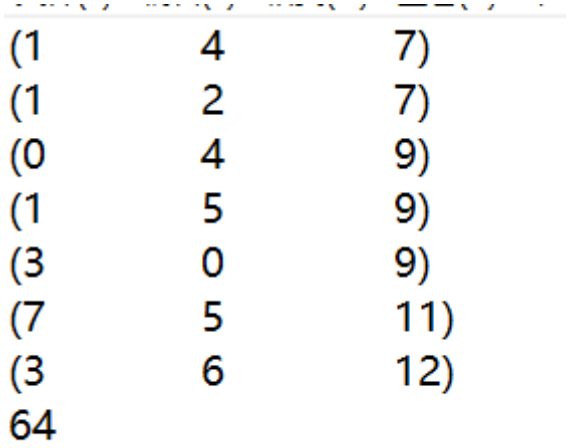
```
            outpath = "../output/result1.txt";
            break;
        case 64:
            path = "../input/input2.txt";
            outpath = "../output/result2.txt";
            break;
        case 128:
            path = "../input/input3.txt";
            outpath = "../output/result3.txt";
            break;
        case 512:
            path = "../input/input4.txt";
            outpath = "../output/result4.txt";
            break;
        default:
            cout << "Input Error!" << endl;
            cout << "Please enter the number of vertices(N:0, 8, 64, 128, 512):" << endl;
            cin >> N;
            continue;
        }

        G.E.clear();    // 初始化
        A.clear();

        Get_Edges(path, &G);
        QueryPerformanceFrequency(&f);
        dqFreq=(double)f.QuadPart;
        QueryPerformanceCounter(&time_start);   //计时开始

        A = MST_KRUSKAL(&G);

        QueryPerformanceCounter(&time_over);    //计时结束
        start = 1e6*time_start.QuadPart/dqFreq;
        finish = 1e6*time_over.QuadPart/dqFreq;
        run_time = 1e6*(time_over.QuadPart-time_start.QuadPart)/dqFreq;//乘以 1e6 把单位由秒化为微秒，精度为1000 000/（cpu主频）微秒

        fprintf(fp_time, "N = %d; start: %lf us; finish: %lf us; run_time: %lf us\n", N, start, finish, run_time); // time.txt

        if((fp_result = fopen(outpath,"w"))==NULL){
            printf("Fail to open result.txt!\n");
            exit(0);
        }
        tree_weight = 0;
        for (i = 0; i < A.size(); i++){
            fprintf(fp_result, "(%d\t%d\t%d)\n", A[i].first.first, A[i].first.second, A[i].second);
            tree_weight += A[i].second;
        }
        fprintf(fp_result, "%d", tree_weight);
        fclose(fp_result);

        cout << "Completed for size " << N << endl;
        cout << "Please enter the number of vertices(N:0, 8, 64, 128, 512):" << endl;
        cin >> N;
```

```
    }
    return(0);
}
```

`main` 函数没有什么难点，主要就是根据输入规模选择路径并进行一些初始化，然后统计运行时间等。

实验结果与分析

规模为 8 的结果如下：



前面几行每行是一条边，每个数值分别代表顶点、顶点和边的权重，最后一行是整棵树(森林)的代价。

每个结点规模对应的运行时间为：

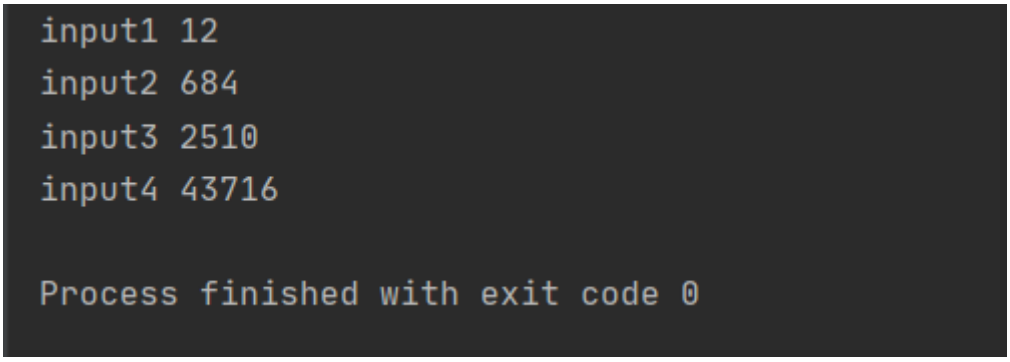
```
N = 8; start: 63422291758.599998 us; finish: 63422291765.299995 us; run_time: 6.700000 us
N = 64; start: 63424178974.300003 us; finish: 63424179133.100006 us; run_time: 158.800000 us
N = 128; start: 63425701401.500008 us; finish: 63425702003.299995 us; run_time: 601.800000 us
N = 512; start: 63429011815.500008 us; finish: 63429026094.300003 us; run_time: 14278.800000 us
```

首先统计每个规模对应的边的数量，代码如下：

```
fp = open("../input/input1.txt", 'r')
text = fp.read()
lines = text.split('\n')
print("input1", len(lines)-1)   # 最后一行为空行
fp.close()

fp = open("../input/input2.txt", 'r')
text = fp.read()
lines = text.split('\n')
print("input2", len(lines)-1)     # 最后一行为空行
fp.close()

fp = open("../input/input3.txt", 'r')
text = fp.read()
lines = text.split('\n')
print("input3", len(lines)-1)     # 最后一行为空行
fp.close()

fp = open("../input/input4.txt", 'r')
```

```
text = fp.read()
lines = text.split('\n')
print("input4", len(lines)-1)      # 最后一行为空行
fp.close()
```

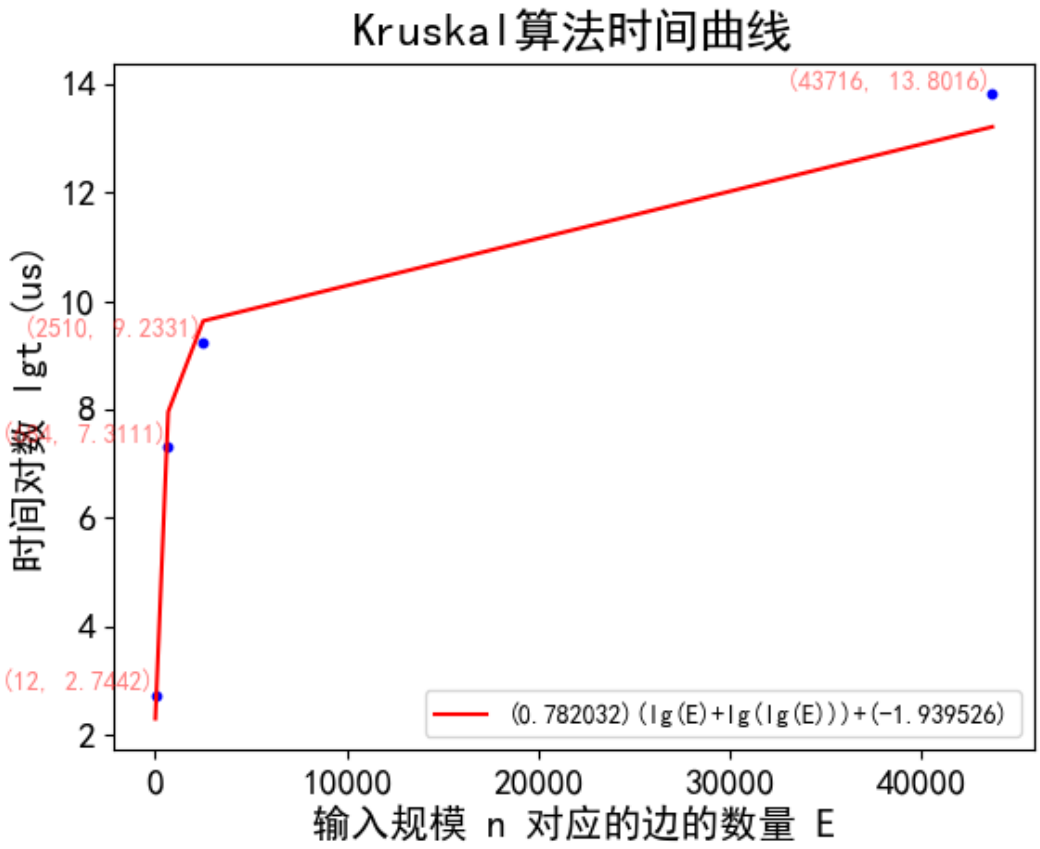
结果如图：



拟合曲线代码：

```
plt.rcParams['font.sans-serif'] = ['SimHei']  # 显示中文标签
plt.rcParams['axes.unicode_minus'] = False  # 这两行需要手动设置


def func(x, a, c):  # 对数函数
    y = a * (np.log2(x) + np.log2(np.log2(x))) + c
    return y


def DrawDiagram(times, path):
    x = np.array([12, 684, 2510, 43716])    # 边数
    y = np.array(times)
    popt, pcov = curve_fit(func, x, y)
    a = round(popt[0], 6)  # 保留6位小数
    c = round(popt[1], 6)
    yvals = func(x, a, c)
    label = '(' + str(a) + ')' + '(' + 'lg(E)' + '+' + 'lg(lg(E))' + ')' + '+' + '(' + str(c) + ')'
    plt.scatter(x, y, s=10, c='blue')  # 将每个规模和对应的运行时间的对数的散点在图中描出来
    for a, b in zip(x, y):
        plt.text(a, b, (a, b), ha='right', va='bottom', fontsize=10, color='r', alpha=0.5)
    plt.plot(x, yvals, c='red', label=label)  # 描绘出拟合曲线
    # plt.plot(x, y, c='blue', label="散点线")  # 描绘出折线图
    plt.legend(loc=4)  # 指定legend图例的位置为右下角
    plt.title("Kruskal算法时间曲线", fontsize=18)  # 标题及字号
    plt.xlabel("输入规模 n 对应的边的数量 E", fontsize=15)  # X轴标题及字号
    plt.ylabel("时间对数 lgt (us)", fontsize=15)  # Y轴标题及字号
    plt.tick_params(axis='both', labelsize=14)  # 刻度大小
    plt.xticks()
    plt.yticks()
    plt.savefig(path)
    plt.show()
```

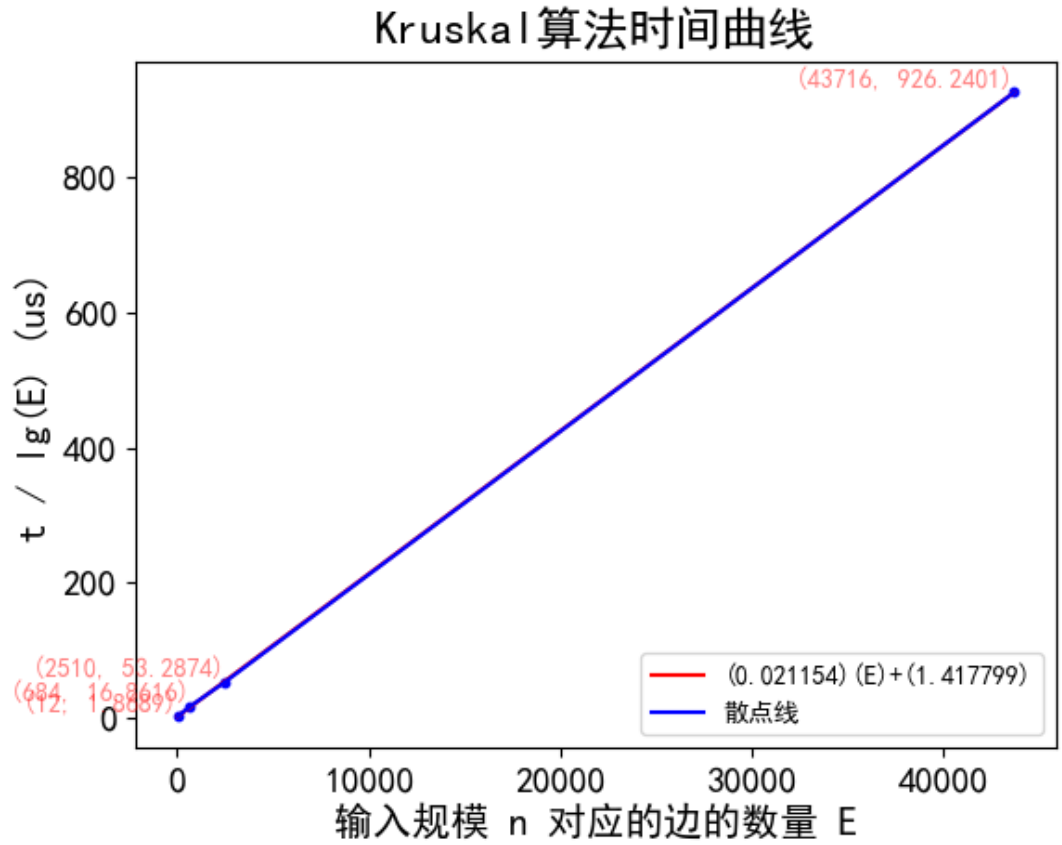
拟合曲线如下：

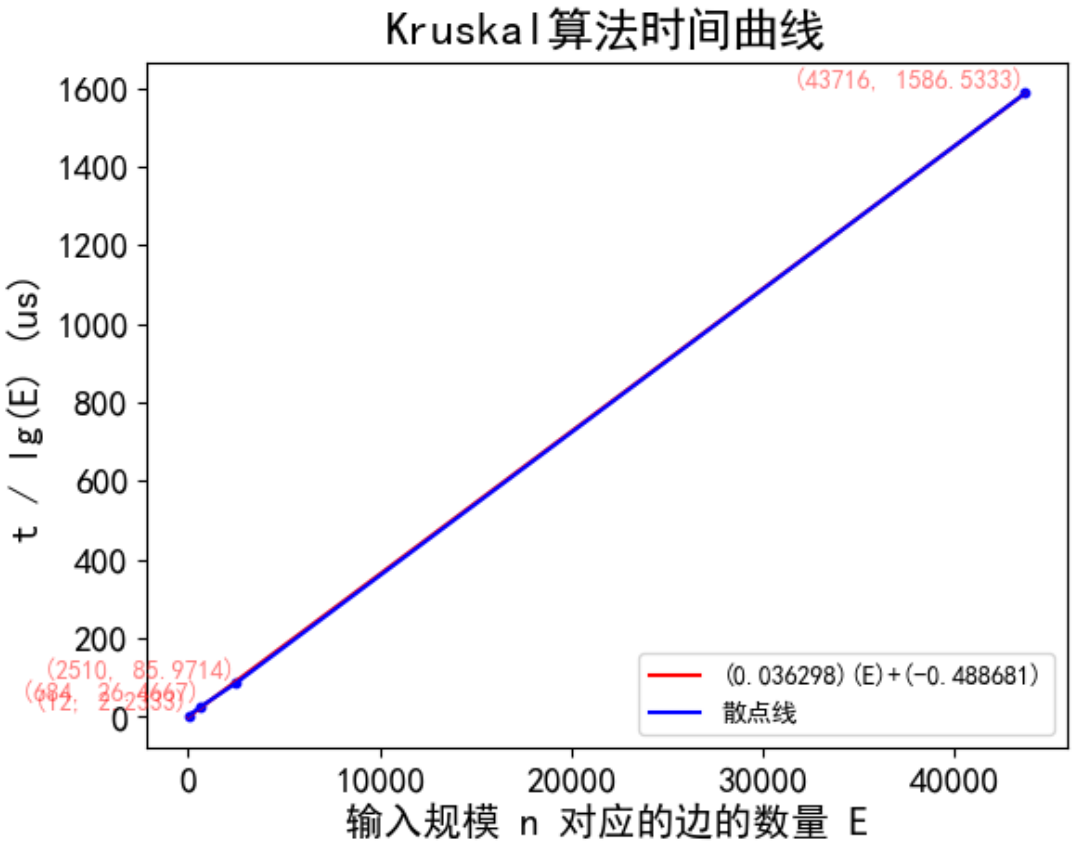


$$
\begin{aligned}
&\text{根据课本可知，} time = O(Elg(E)) \\
&\text{即} \quad \exists c_1 > 0, \text{使得} t \leq c_1 Elg(E) \\
&lg(t) \leq lgc_1 + lg(E) + lg(lg(E))
\end{aligned}
$$

拟合出来的系数为`0.782032`，和理论分析的结果很接近，因此可以认为$time = O(ElgE)$. 由于$|E| \leq |V|^2$，因此也可以认为$time = O(ElgV)$.由此可以认为该算法复杂度与课本中的理论复杂度是相同的。

为了让图更直观，绘出$E - \frac{t}{lgE}$和$E - \frac{t}{lgV}$的图像：





蓝色的线是折线图，红色的线是拟合曲线，两者很相似，都近似为直线，即E与$\frac{t}{lgE}$成正相关，E与$\frac{t}{lgV}$成正相关，符合课本的复杂度。

实验4.2 Johnson算法

实验内容及要求

■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4.2 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

实验方法与步骤

- 首先随机生成图(有向图，两种边的规模 $log_5 N$和$log_7 N$):

```
vector<pair<int, int> >E;     // 边集合
vector<int> weights;          // 权重

void Rand_Input(int N, const char *path, int base){     // base表示基底，为5或7， N 表示结点数， path为输出路径
     pair<int, int> rand_edge;     // 随机生成的边
     int weight;
     FILE* fp;
    srand((unsigned)time(NULL));     // 用系统定时/计数器的值作为随机种子，产生比较好的随机数
    for(int i=0; i<N; i++){     // 为每个结点生成指定的边
        rand_edge.first = i;
        int edge_number = int(log(N) / log(base));     // 计算边的数量
        int j = 0;
        while (j < edge_number){     // 为有向图，因此生成 edge_number 条边即可
            rand_edge.second = rand() % N;     // 连结的另外一个结点
            if(find(E.begin(), E.end(), rand_edge) == E.end()){  // 查看是否与以前生成的边重复,注意这里是有向图
                weight = rand() % 51;       // 边的权重为 [0, 50]
                E.push_back(rand_edge);          // 加入这条边
                weights.push_back(weight);
                j++;
            }
```

```
        }
    }
    if((fp=fopen(path,"w"))==NULL){          // 输入
        printf("Fail to open file!\n");
        exit(0);
    }
    for(int k = 0; k < E.size(); k++ ){
        fprintf(fp,"%d\t%d\t%d\n",E[k].first, E[k].second, weights[k]);
        printf("%d\n",k+1);
    }
    E.clear();  // 清空边
    weights.clear();    // 清空权重
    printf("OK!!!\n");
    fclose(fp);
    return;
}
```

随机生成[0, 50]范围的整数边。

- Johnson算法的数据结构：

```
#define MAX_WEIGHT 10000000     // 权重最大值
#define MAX 1000                // 顶点数最大值
#define PARENT(i) (i-1)/2       //父节点 用于构建堆
#define LEFT(i) 2*i+1           //左孩子
#define RIGHT(i) 2*i+2          //右孩子

int D[MAX][MAX];    // 所有结点对的最短路径距离
int Pre[MAX][MAX];  // 记录前驱
bool no_negative_weight_cycle = true;   // 标志是否有负环

typedef struct vertex{
    int num;        // num存储顶点编号
    int d;          // 最短路径估计值
    int id;       // 用于DIJKSTRA算法中，记录次序
    vertex *p;      // 前驱顶点
    vector<pair<int, int> >edges; // 邻接表,第一个int为顶点编号,也对应着在图G.V容器中的存储下标;第二个int为边的权值
}Vertex;

typedef struct graph{   // 有向图
    vector<Vertex *>V;  // 图结构中，存储结点的容器，表示顶点集合
}Graph;
```

- `RELAX`函数，用于松弛顶点的最短距离估计`d`

```
void RELAX(Vertex *u, Vertex *v, int w){
    if (v->d > u->d + w){
        v->d = u->d + w;
        v->p = u;
    }
    return;
}
```

- 初始化函数：

```
void INITIALIZE_SINGLE_SOURCE(Graph &G, Vertex *s){    // Initialization
    for (auto v : G.V){
        v->d = MAX_WEIGHT;
        v->p = nullptr;
    }
    s->d = 0;
}
```

- `BELLMAN_FORD`函数：

```
bool BELLMAN_FORD(Graph &G, Vertex *s){
    int i;
    INITIALIZE_SINGLE_SOURCE(G, s);
    for (i = 1; i < G.V.size(); i++)
        for (auto u : G.V)              // 相比于课本而言多了一个for，这是因为采用邻接表存储，要遍历所有顶点才能获得所有边
            for (auto edge : u->edges)
                RELAX(u, G.V[edge.first], edge.second);
    for (auto u : G.V)
        for (auto edge : u->edges)
            if (G.V[edge.first]->d > u->d + edge.second)
                return false;
    return true;
}
```

用于检查图中是否含有负环，算法和课本一致。

由于`DIJKSTRA`算法要用到最小堆，因此首先需要实现最小堆。最小堆相关算法如下：

```
void MIN_HEAPIFY(vector<Vertex *> &A, int i, int heapsize){ // 假设左右子树已经有序，将当前节点放入指定位置
    int l, r, min;
    l = LEFT(i);
    r = RIGHT(i);
    if(l < heapsize && A[l]->d < A[i]->d)
        min = l;
    else
        min = i;
    if(r < heapsize && A[r]->d < A[min]->d)
        min = r;
    if(min != i){
        auto x = A[i];
        A[i] = A[min];
        A[i]->id = i;
        A[min] = x;
        A[min]->id = min;
        MIN_HEAPIFY(A, min, heapsize);
    }
}
```

```
20
21  Vertex *HEAP_EXTRACT_MIN(vector<Vertex *> &A, int heapsize){    // 获得最小值
    时间复杂度 O(lgn)
22      if(heapsize < 1){
23          printf("heap underflow! \n");
24          return(nullptr);
25      }
26      Vertex *min = A[0];
27      A[0] = A[heapsize-1];
28      heapsize--;
29      MIN_HEAPIFY(A, 0, heapsize);
30      return min;
31  }
32
33  void HEAP_DECREASE_KEY(vector<Vertex *> &A, int i, int key){    // 时间复杂度
    O(lgn)
34      if(key > A[i]-> d){
35          printf("new key is larger than current key!\n");
36          return;
37      }
38      A[i]->d = key;
39      while(i > 0 & A[PARENT(i)]->d > A[i]->d){
40          auto x = A[i];
41          A[i] = A[PARENT(i)];
42          A[i]->id = i;
43          A[PARENT(i)] = x;
44          A[PARENT(i)]->id = PARENT(i);
45          i = PARENT(i);
46      }
47  }
48
49  void BUILD_MIN_HEAP(vector<Vertex *> &A, int heapsize){     // 建堆 O(n)
50      for(int i = heapsize/2 - 1; i >= 0; i--)
51          MIN_HEAPIFY(A, i, heapsize);
52  }
53
```

算法是根据第6章堆排序实现的(优先队列和最小堆)，在`DIJKSTRA`算法中，由于要获得一个具体的结点在最小堆里的位置，因此需要维护一个数据`id`,用于表示该结点在最小堆里的下标，从而使得能在`O(1)`的时间内在最小堆中找到该结点。

- `DIJKSTRA`算法：

```
1   void DIJKSTRA(Graph &G, Vertex *s){
2       Vertex *v, *u;
3       INITIALIZE_SINGLE_SOURCE(G, s);      // 初始化
4       vector<Vertex *>Q;
5       int i = 0;
6       for (auto v : G.V) {
7           v->id = i++;         // 在最小堆里的下标
8           Q.push_back(v);
9       }
10      BUILD_MIN_HEAP(Q, G.V.size());       // 建堆
11      while (!Q.empty()){
12          u = HEAP_EXTRACT_MIN(Q, Q.size());  // 获得距离最近结点
```

```
        Q.pop_back();   //弹出最后一个元素以更新维护Q.size()
        for (auto edge : u->edges){
            if(G.V[edge.first]->d > u->d + edge.second){
                HEAP_DECREASE_KEY(Q, G.V[edge.first]->id, u->d +
edge.second);      // 更新最小距离的估计
                G.V[edge.first]->p = u;     // 更新前驱结点
            }
        }
    }
    return;
}
```

算法根据课本伪代码实现，讲解可见注释。

- `JOHNSON`算法：

```
void JOHNSON(Graph G){
    int *h;
    Vertex *s = new Vertex;
    Graph G1 = G;
    s->num = G1.V.size();       // 构建s结点
    for (auto v : G1.V){
        s->edges.push_back(pair<int, int>(v->num, 0));
    }
    G1.V.push_back(s);
    if (BELLMAN_FORD(G1, s) == false){
        cout << "The input graph contains a negative-weight cycle." << endl;
        no_negative_weight_cycle = false;   // 有负环
    }
    else{
        h = (int *)malloc(sizeof(int)*G1.V.size());     // 构建 h 函数
        for(auto v : G1.V)
            h[v->num] = v->d;
        for(auto v : G1.V)  // 对于指针，这样引用即可，会改变G1里的具体值
            for (auto edge = v->edges.begin(); edge < v->edges.end();
edge++)
                (*edge).second = (*edge).second + h[v->num] -
h[(*edge).first]; // 构建新的权值函数
        for(auto u : G1.V){
            DIJKSTRA(G1, u);
            for(auto v : G1.V){
                D[u->num][v->num] = v->d + h[v->num] - h[u->num];       //
记录值
                if(v->p == nullptr){
                    Pre[u->num][v->num] = -1;       // 标记源或者到达不了
                }
                else{
                    Pre[u->num][v->num] = v->p->num;    // 标记前驱
                }
            }
        }
    }
    return;
}
```

其中，二维数组`D`中的任意一个元素`D[i][j]`代表的是结点`i`到结点`j`的最短距离； 二维数组`Pre`中的任意一个元素`Pre[i][j]`代表的是结点`i`到结点`j`的最短路径中`j`的前驱。

- 下面是一些初始化函数：

```
void MAKE_SET(Graph &G, int N) {    // 生成N个顶点
    int i;
    Vertex *node;
    for (i = 0; i < N; i++) {
        node = new Vertex;
        node->num = i;
        node->id = i;       // 用于DIJKSTRA算法中, 记录次序
        G.V.push_back(node);
    }
    return;
}

void Get_Edges(const char *path, Graph &G){     // 构建邻接表
    int i;
    FILE *fp;
    pair<pair<int, int>, int>edge;    // 边
    if((fp=fopen(path,"r"))==NULL){
        printf("Fail to open file!\n");
        exit(0);
    }
    while(fscanf(fp, "%d\t%d\t%d", &edge.first.first, &edge.first.second,
&edge.second) != EOF){
        G.V[edge.first.first]->edges.push_back(pair<int, int>
(edge.first.second, edge.second));
    }
    fclose(fp);
}
```

- 输出函数：

```
void Output(int N, const char *path){ //  N 个结点
    string route;
    int k;
    FILE *fp;
    if((fp=fopen(path,"w"))==NULL){
        printf("Fail to open file!\n");
        exit(0);
    }
    for(int i = 0; i < N; i++){
        for(int j = 0; j< N; j++){
            if(Pre[i][j] == -1 && j != i){      // 到达不了,或者是相同结点
                fprintf(fp, "( %d -> %d can't reach)\n", i, j);
            }
            else if(Pre[i][j] != -1){
                route = to_string(D[i][j]) + ")\n";
                k = j;
                while(Pre[i][k] != -1){
                    route = to_string(k) + ", " + route;
```

```
                    k = Pre[i][k];      // 得到前驱
                }
                route = "( " + to_string(k) + ", " + route;
                fprintf(fp,"%s", route.c_str());
            }
        }
    }
    fclose(fp);
}
```

- `main`函数:

```
int main(){
    int i;
    int N, base;    // 结点数和基底
    const char *path, *outpath;
    Graph G;        // 图
    FILE *fp_time;

    double run_time, start, finish;
    _LARGE_INTEGER time_start;  //开始时间
    _LARGE_INTEGER time_over;   //结束时间
    double dqFreq;      //计时器频率
    LARGE_INTEGER f;    //计时器频率

    cout << "Please enter the number of vertices(N:0(quit), 27, 81, 243, 729) and base(5, 7):" << endl;
    cin >> N >> base;

    if((fp_time = fopen("../output/time.txt","w"))==NULL){
        printf("Fail to open time.txt!\n");
        exit(0);
    }

    while(N){
        switch (N){
        case 27:
            if(base == 5){
                path = "../input/input11.txt";
                outpath = "../output/result11.txt";
            }
            else if(base == 7){
                path = "../input/input12.txt";
                outpath = "../output/result12.txt";
            }
            else{
                cout << "Input Error!" << endl;
                cout << "Please enter the number of vertices(N:0(quit), 27, 81, 243, 729) and base(5, 7):" << endl;
                cin >> N >> base;
                continue;
            }
            break;
        case 81:
            ...
```

```
            break;
        case 243:
            ...
            break;
        case 729:
            ...
            break;
        default:
            cout << "Input Error!" << endl;
            cout << "Please enter the number of vertices(N:0(quit), 27, 81, 243, 729) and base(5, 7):" << endl;
            cin >> N >> base;
            continue;
        }

        G.V.clear();    // 初始化

        MAKE_SET(G, N);     // 生成 N 个顶点
        Get_Edges(path, G); // 获得边

        QueryPerformanceFrequency(&f);
        dqFreq=(double)f.QuadPart;
        QueryPerformanceCounter(&time_start);   //计时开始

        JOHNSON(G);

        QueryPerformanceCounter(&time_over);    //计时结束
        start = 1e6*time_start.QuadPart/dqFreq;
        finish = 1e6*time_over.QuadPart/dqFreq;
        run_time = 1e6*(time_over.QuadPart-time_start.QuadPart)/dqFreq;//乘以 1e6 把单位由秒化为微秒，精度为1000 000/（cpu主频）微秒

        if(no_negative_weight_cycle){   // 无负环时输出
            fprintf(fp_time, "N = %d; base = %d; start: %lf us; finish: %lf us; run_time: %lf us\n", N, base, start, finish, run_time); // time.txt
            Output(N, outpath);     // 输出
            cout << "Completed for size " << N << " and base " << base << endl;
        }

        no_negative_weight_cycle = true;    // 开始新的一轮判断
        cout << "Please enter the number of vertices(N:0(quit), 27, 81, 243, 729) and base(5, 7):" << endl;
        cin >> N >> base;
    }
    fclose(fp_time);
    return(0);
}

```

实验结果与分析

规模为 27 的base=5部分结果如下：

```
( 0, 20, 25, 19, 24, 1, 129)
( 0 -> 2 can't reach)
( 0, 20, 25, 19, 24, 26, 3, 136)
( 0, 20, 16, 4, 132)
( 0 -> 5 can't reach)
( 0 -> 6 can't reach)
( 0, 11, 7, 82)
( 0, 20, 25, 19, 24, 26, 3, 8, 162)
( 0, 20, 25, 19, 24, 26, 3, 8, 22, 9, 219)
( 0, 20, 25, 19, 10, 127)
( 0, 11, 46)
( 0 -> 12 can't reach)
( 0 -> 13 can't reach)
( 0, 20, 16, 4, 14, 132)
( 0, 20, 25, 19, 24, 26, 15, 167)
( 0, 20, 16, 88)
( 0, 20, 16, 17, 118)
( 0, 11, 7, 18, 109)
( 0, 20, 25, 19, 80)
( 0, 20, 43)
( 0, 11, 7, 18, 21, 131)
( 0, 20, 25, 19, 24, 26, 3, 8, 22, 196)
( 0 -> 23 can't reach)
( 0, 20, 25, 19, 24, 118)
( 0, 20, 25, 78)
( 0, 20, 25, 19, 24, 26, 126)
( 1 -> 0 can't reach)
( 1 -> 2 can't reach)
( 1, 10, 3, 55)
( 1, 10, 3, 20, 16, 4, 154)
```

每行代表的是一条路径，每行的最后一个数值代表的是这条路径的代价(花费)，倒数第二个数值代表的是终点，第一个数值代表起点。

每个结点规模对应的运行时间为：

```
N = 27; base = 5; start: 163587092800.199980 us; finish: 163587093203.600010 us; run_time: 403.400000 us
N = 27; base = 7; start: 163589922717.799990 us; finish: 163589923171.399990 us; run_time: 453.600000 us
N = 81; base = 5; start: 163592957264.300020 us; finish: 163592960007.299990 us; run_time: 2743.000000 us
N = 81; base = 7; start: 163595592271.299990 us; finish: 163595594709.799990 us; run_time: 2438.500000 us
N = 243; base = 5; start: 163598721965.699980 us; finish: 163598751358.000000 us; run_time: 29392.300000 us
N = 243; base = 7; start: 163601912703.100010 us; finish: 163601936151.699980 us; run_time: 23448.600000 us
N = 729; base = 5; start: 163605448957.500000 us; finish: 163605737342.599980 us; run_time: 288385.100000 us
N = 729; base = 7; start: 163613427072.899990 us; finish: 163613693947.800020 us; run_time: 266874.900000 us
```

每个规模对应的边的数量为：

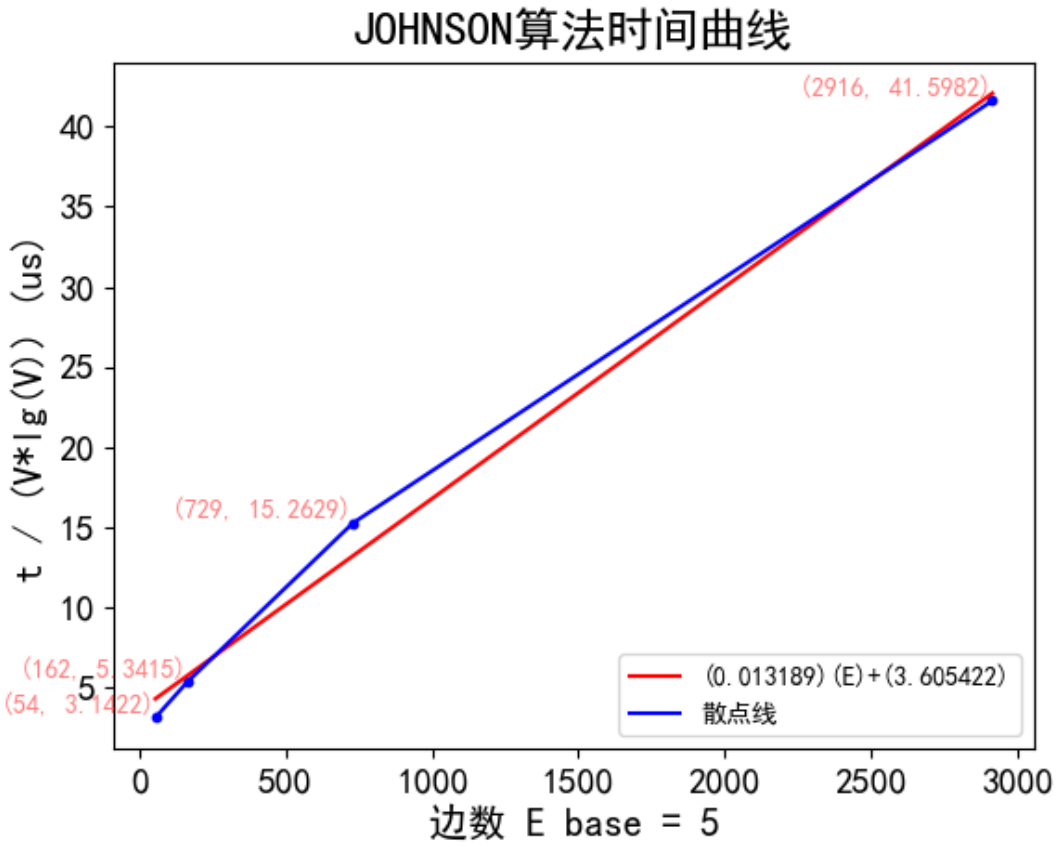
$$
\begin{aligned}
N=27, base=5, edges\_number = N*log_5 N = 54\\
N=27, base=7, edges\_number = N*log_7 N = 27\\
N=81, base=5, edges\_number = N*log_5 N = 162\\
N=81, base=7, edges\_number = N*log_7 N = 162\\
N=243, base=5, edges\_number = N*log_5 N = 729\\
N=243, base=7, edges\_number = N*log_7 N = 486\\
N=729, base=5, edges\_number = N*log_5 N = 2916\\
N=729, base=7, edges\_number = N*log_7 N = 2187
\end{aligned}
$$

拟合曲线代码：

```python
def func(x, a, c):  # 对数函数
    y = a * x + c
    return y


def DrawDiagram(times, path):
    x = np.array([54, 162, 729, 2916])
    y = np.array(times)
    popt, pcov = curve_fit(func, x, y)  # 最小二乘法拟合
    a = round(popt[0], 6)  # 保留6位小数
    c = round(popt[1], 6)
    yvals = func(x, a, c)
    label = '(' + str(a) + ')' + '(' + 'E' + ')' + '+' + '(' + str(c) + ')'
    plt.scatter(x, y, s=10, c='blue')  # 将每个规模和对应的运行时间的对数的散点在图中描出来
    for a, b in zip(x, y):
        plt.text(a, b, (a, b), ha='right', va='bottom', fontsize=10, color='r', alpha=0.5)
    plt.plot(x, yvals, c='red', label=label)  # 描绘出拟合曲线
    plt.plot(x, y, c='blue', label="散点线")  # 描绘出折线图
    plt.legend(loc=4)  # 指定legend图例的位置为右下角
    plt.title("JOHNSON算法时间曲线", fontsize=18)  # 标题及字号
    plt.xlabel("边数 E base = 5", fontsize=15)  # X轴标题及字号
    plt.ylabel("t / (V*lg(V)) (us)", fontsize=15)  # Y轴标题及字号
    plt.tick_params(axis='both', labelsize=14)  # 刻度大小
    plt.xticks()
    plt.yticks()
    plt.savefig(path)
    plt.show()


def main():
    times = [403.4, 2743, 29392.3, 288385.1]
    times[0] = round(times[0] / (27 * math.log(27, 2)), 4)
    times[1] = round(times[1] / (81 * math.log(81, 2)), 4)
    times[2] = round(times[2] / (243 * math.log(243, 2)), 4)
```

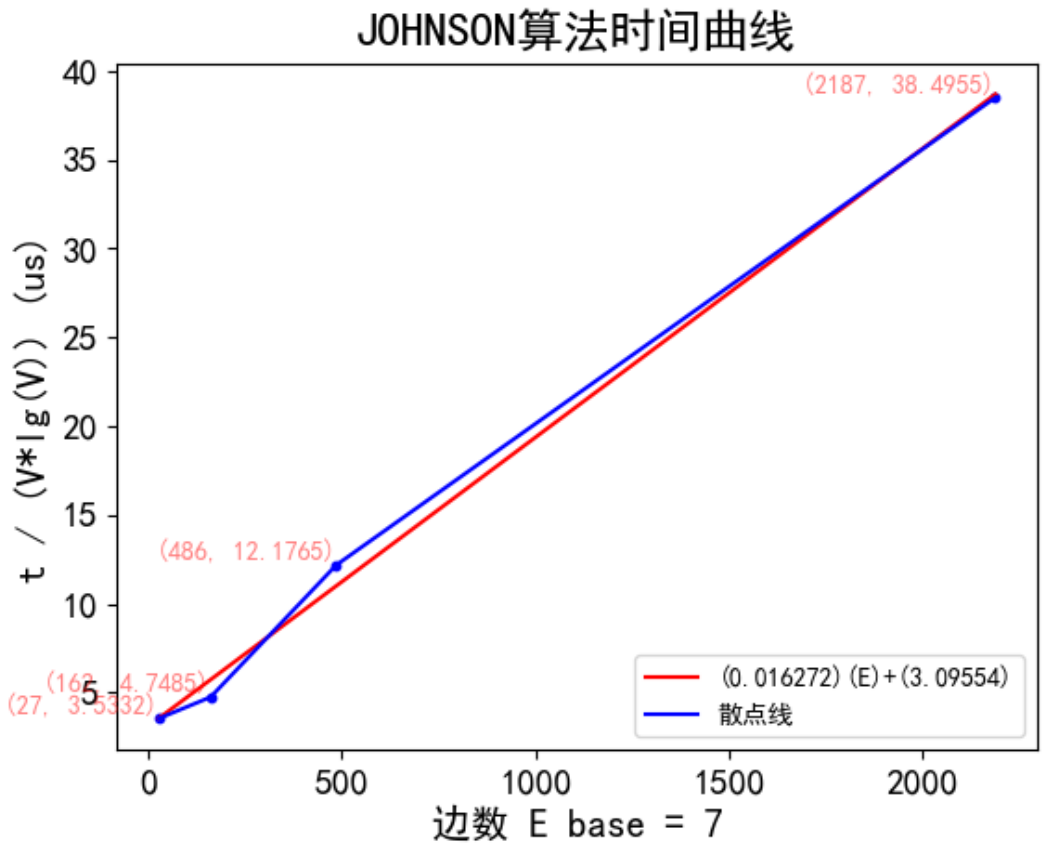
```
    times[3] = round(times[3] / (729 * math.log(729, 2)), 4)
    path = r'C:\Desktop\57-曾勇程-PB18000268-project4\figs\Jtime1.png'
    DrawDiagram(times, path)
```

$base = 5$ 的拟合曲线如下：



蓝色的线是折线图，红色的线是拟合曲线，可以看出两者很相似，说明 E 和 $\frac{t}{VlgV}$ 成正相关，符合课本的$T(n) = O(VElgV)$的时间复杂度描述。

$base = 7$ 的拟合曲线如下：



蓝色的线是折线图，红色的线是拟合曲线，可以看出两者很相似，说明 E 和 $\frac{t}{VlgV}$ 成正相关，符合课本的$T(n) = O(VElgV)$的时间复杂度描述。