# 中国科学技术大学计算机学院

# 算法基础实验报告

## 实验四

## 图算法


学　　号：　PB18000203

姓　　名：　　汪洪韬

专　　业：计算机科学与技术

指导老师：　　顾乃杰

中国科学技术大学计算机学院


2021 年 1 月 6 日

# 一、 实验内容

1. 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。

2. 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。）

# 二、 实验设备和环境

1.实验设备：PC 机；

2.实验环境：Visual Studio 2019

# 三、 实验方法和步骤

## 1. Kurskal 算法

图的数据结构

```
typedef struct arc {
    int w;
    int i;
    int j;
    int rank;
    friend bool operator <(arc node1, arc node2)//overloaded function
    {
        return node1.w > node2.w;
    }
    friend bool operator >(arc node1, arc node2)
    {
        return node1.w > node2.w;
    }
}arc;//arc in Graph


typedef struct ver {
    int rank;
    int info;
    int r;
    struct ver* p;
}ver;// vertice node

typedef struct Graph {
    int vernum;
    int arcnum;
    struct ver* ver;
    struct arc* arc;
}Graph;// Graph
```

## 随机生成边与权值

```c
int generatearc(int scale, FILE* fp) {//随机生成边与权值
    int* ver_arcnum = (int*)malloc(sizeof(int) * (scale + 5));
    bool* verarc = (bool*)malloc(sizeof(int) * (scale * scale / 2 + 5));
    int i, j, w , rdm, sum = 0;
    for (i = 0; i < scale * scale / 2; i++)
        *(verarc + i) = 1;
    for (i = 0; i < scale; i++)
        ver_arcnum[i] = 0;
    for (i = 0; i < scale; i++) {
        rdm = 1 + rand() % (scale / 2) - ver_arcnum[i];
        while (rdm > 0) {
            j = rand() % scale;
            if (ver_arcnum[j] < scale / 2 && *(verarc + i * (scale / 2) + j)) {
                w = 1 + rand() % 20;
                *(verarc + i * (scale / 2) + j) = 0;
                *(verarc + j * (scale / 2) + i) = 0;
                fprintf(fp, "%d %d %d\n", i, j, w);
                sum++;
                rdm--;
                ver_arcnum[i]++;
                if (i != j)
                    ver_arcnum[j]++;
            }
        }
    }
    free(ver_arcnum);
    free(verarc);
    return sum;
}
```

## 构建无向图（有自环无重边）

```c
void buildGraph(Graph *G, int scale, FILE* fp) {//构建无向图
    G->ver = (ver*)malloc(sizeof(ver) * (scale + 5));
    G->arc = (arc*)malloc(sizeof(arc) * (G->arcnum + 5));
    ver* p;
```

```
    int i, j, w, k;
    for (k = 0; k < scale; k++) {
        G->ver[k].rank = k;
        G->ver[k].info = k;
        G->ver[k].p = NULL;
    }// make vertice node
    for (k = 0; k < G->arcnum; k++) {
        fscanf(fp, "%d %d %d", &i, &j, &w);
        G->arc[k].w = w;
        G->arc[k].i = i;
        G->arc[k].j = j;
        G->arc[k].rank = k;
    }
}
```

## 对不相交集合的操作

```
void makeSet(ver *v) {//新建集合
    v->p = v;
    v->r = 0;
}

ver *findSet(ver *v) {//找到集合的根节点
    if (v != v->p)
        v->p = findSet(v->p);
    return v->p;
}

void linkSet(ver *u, ver *v) {//链接两个集合
    if (u->r > v->r)
        v->p = u;
    else {
        u->p = v;
        if (u->r == v->r)
            v->r++;
    }
}

```

```
void unionSet(ver *u, ver *v) {//将 u,v 所在的集合合并
    linkSet(findSet(u), findSet(v));
}
```

## Kruskal 算法

```
void Kruscal(Graph *G) {
    int k, x, y, curarc, innum = 0;
    cost = 0;
    for (k = 0; k < MAXIMUM; k++)
        arcin[k] = 0;
    priority_queue<arc, vector<arc>, less<arc>> pque;//实现以边权值为比较标准的优先队列
    for (k = 0; k < G->vernum; k++) {
        makeSet(&G->ver[k]);
    }
    for (k = 0; k < G->arcnum; k++)
        pque.push(G->arc[k]);
    for (k = 0; k < G->arcnum; k++) {
        x = pque.top().i;
        y = pque.top().j;
        curarc = pque.top().rank;
        pque.pop();//弹出当前权值最小的边
        if (findSet(&G->ver[x]) != findSet(&G->ver[y])) {//若此边连接了两个不相交的集合，则将它加入最小生成树的边集
            arcin[innum++] = curarc;
            cost += G->arc[curarc].w;
            unionSet(&G->ver[x], &G->ver[y]);
        }
    }
}
```

## 打印最小生成树

```c
void printGraph(Graph *G, int scale, FILE* fp) {
    int k;
    fprintf(fp,"the whole cost is %d\n", cost);
    for (k = 0; k < scale - 1; k++)
        fprintf(fp, "%d %d %d\n", G->arc[arcin[k]].i, G->arc[arcin[k]].j, G->arc[arcin[k]].w);
}
```

## 主函数

```c
int main() {
    double run_time[NUM];
    _LARGE_INTEGER time_start;
    _LARGE_INTEGER time_over;
    double dqFreq;
    LARGE_INTEGER f;
    QueryPerformanceFrequency(&f);
    dqFreq = (double)f.QuadPart;//timing module

    FILE* fpin, * fpout, * fptime;
    fptime = fopen("../output/time.txt", "w+");
    int timeloop = 0, loop = 0, sum;
    for (loop = 0; loop < NUM; loop++) {
            fpin = fopen(filein[loop], "w+");
            fpout = fopen(fileout[loop], "w+");
            if (fpin == NULL || fpout == NULL) {
                printf("Can't open the files!\n");
                exit(0);
            }//file pointers
            sum = generatearc(scale[loop], fpin);
            fseek(fpin, 0, SEEK_SET);
            Graph *G;
            G = (Graph*)malloc(sizeof(Graph));
            G->arcnum = sum;
            G->vernum = scale[loop];
            buildGraph(G, scale[loop], fpin);
            QueryPerformanceCounter(&time_start);//start timing
```

```
            Kruscal(G);
            QueryPerformanceCounter(&time_over);//end timing
            printGraph(G, scale[loop], fpout);
            run_time[loop] = 1000000 * (time_over.QuadPart - time_start.QuadPart
) / dqFreq;//caculate the running time
            fclose(fpin);
            fclose(fpout);
        }
    for (loop = 0; loop < NUM; loop++)
        fprintf(fptime, "%d data's inserting cost %lfus\n", scale[loop], run_tim
e[loop]);

    fclose(fptime);
    return 0;
}
```

## 2. Johnson 算法

有向图的数据结构

```
typedef struct arc {
    int w;
    int i;
    int j;
    int rank;
}arc;//arc in Graph

typedef struct ver {
    int rank;
    int info;
    int d;
    struct ver* pi;
    struct ver* p;
    int adj[10];
    int adjnum;
    friend bool operator <(ver node1, ver node2)// overloaded function to compar
e
    {
        return node1.d > node2.d;
```

```
    }
    friend bool operator >(ver node1, ver node2)
    {
        return node1.d > node2.d;
    }
}ver;// vertice node

typedef struct Graph {
    int vernum;
    int arcnum;
    struct ver* ver;
    struct arc* arc;
}Graph;// Graph
```

## 随机生成有向边及其权值

```
void generatearc(int scale, int arcscale, FILE* fp) {//随机生成边与权值
    bool* verarc = (bool*)malloc(sizeof(int) * (scale * scale / 2 + 5));
    int i, j, w, rdm;
    fprintf(fp, "%d %d\n", scale, arcscale);
    for (i = 0; i < scale * scale / 2; i++)
        *(verarc + i) = 1;
    for (i = 0; i < scale; i++) {
        rdm = arcscale;
        while (rdm > 0) {
            j = rand() % scale;
            if (*(verarc + i * (scale / 2) + j)) {
                w = rand() % 51;
                *(verarc + i * (scale / 2) + j) = 0;
                fprintf(fp, "%d %d %d\n", i, j, w);
                rdm--;
            }
        }
    }
    free(
```

## 构建有向图

```
void buildGraph(Graph* G, int scale, FILE* fp) {//构建有向图
```

```c
    G->ver = (ver*)malloc(sizeof(ver) * (scale + 10));
    G->arc = (arc*)malloc(sizeof(arc) * (G->arcnum * 2 + 5));
    int i, j, w, k;
    fscanf(fp, "%d %d", &i, &j);
    for (k = 0; k < scale; k++) {
        G->ver[k].rank = k;
        G->ver[k].info = k;
        G->ver[k].p = NULL;
        G->ver[k].adjnum = 0;
    }// make vertice node
    for (k = 0; k < G->arcnum; k++) {
        fscanf(fp, "%d %d %d", &i, &j, &w);
        G->arc[k].w = w;
        G->arc[k].i = i;
        G->arc[k].j = j;
        G->arc[k].rank = k;
        G->ver[G->arc[k].i].adj[G->ver[G->arc[k].i].adjnum++] = j;
    }
}
```

## 单源最短路径操作

```c
void initialSingle(Graph* G, ver *s) {//单源最短路径的初始化
    int k;
    for (k = 0; k < G->vernum; k++) {
        G->ver[k].d = MAX;
        G->ver[k].pi = NULL;
    }
    s->d = 0;
}

void relax(ver* u, ver* v, int** w) {//松弛操作
    if (v->d > u->d + w[u->rank][v->rank]) {
        v->d = u->d + w[u->rank][v->rank];
        v->pi = u;
    }
}
```

## Bellman-Ford 算法

```
int BellmanFord(Graph* G, ver *s, int **w) {//Bellman-Ford 算法，对每条边进行 G.V-1 次松弛操作
    int k,l;
    initialSingle(G, s);
    for (k = 1; k < G->vernum; k++)
        for (l = 0; l < G->arcnum; l++)
            relax(&G->ver[G->arc[l].i], &G->ver[G->arc[l].j], w);
    for (k = 0; k < G->arcnum; k++)
        if (G->ver[G->arc[k].j].d > G->ver[G->arc[k].i].d + w[G->ver[G->arc[k].i].rank][G->ver[G->arc[k].j].rank])
            return 0;
    return 1;
}
```

## Dijkstra 算法

```
void Dijkstra(Graph* G, ver* s, int **w) {//Dijkstra 算法
    int k, l, u, a;
    initialSingle(G, s);
    bool* S;
    S = (bool*)malloc(sizeof(int) * (G->vernum + 5));
    for (k = 0; k < G->vernum; k++) {
        S[k] = 1;
    }
    l = G->vernum;
    while (l--) {
        priority_queue<ver, vector<ver>, less<ver>> pque;//优先队列实现以 v.d 为标准的排序
        for (k = 0; k < G->vernum; k++)
            if (S[k]) {
                pque.push(G->ver[k]);
            }
        u = pque.top().rank;//每次弹出 v.d 最小的一个顶点
        S[u] = 0;
        for (a = G->ver[u].adjnum; a > 0; a--) {
            relax(&G->ver[u], &G->ver[G->ver[u].adj[a - 1]], w);
        }
```

```
        while (!pque.empty())
            pque.pop();
    }
    free(S);
}
```

## 打印路径

```
void printGraph(int s, Graph* G, FILE* fp) {//打印路径
    int k, t;
    int* trace;
    trace = (int*)malloc(sizeof(int) * (G->vernum + 5));
    ver* p;
    for (k = 0; k < G->vernum; k++) {
        if (k == s) {
            ;
        }
        else if (G->ver[k].pi == NULL)
            fprintf(fp, "(%d, %d no exist)\n", s, G->ver[k].rank);
        else {
            p = &G->ver[k];
            t = 0;
            while (p->rank != s) {
                trace[t++] = p->rank;
                p = p->pi;
            }
            trace[t] = s;
            fprintf(fp, "(");
            for (t = t ; t > 0; t--)
                fprintf(fp, "%d,", trace[t]);
            fprintf(fp, "%d %d)\n", trace[0], G->ver[k].d);
        }
    }
}
```

## Johnson 算法

```
int* Johnson(Graph* G, FILE*fp) {//Johnson 算法
    int k, l, s;
```

```c
    int** win, ** win2;

    s = G->vernum;
    Graph *G2=(Graph*)malloc(sizeof(Graph));

    G2->arcnum = G->arcnum;
    G2->vernum = G->vernum;

    G2->ver = (ver*)malloc(sizeof(ver) * (G->vernum + 10));
    G2->arc = (arc*)malloc(sizeof(arc) * (G->arcnum * 2 + 5));
    memcpy(G2->ver, G->ver, sizeof(ver) * (G->vernum + 10));
    memcpy(G2->arc, G->arc, sizeof(arc) * (G->arcnum * 2 + 5));
    G2->ver[G2->vernum].info = s;
    G2->ver[G2->vernum].rank = s;
    G2->vernum++;

    for (k = G2->arcnum; k < G2->arcnum + G2->vernum - 1; k++) {
        G2->arc[k].i = s;
        G2->arc[k].j = k - G2->arcnum;
        G2->arc[k].w = 0;
        G2->arc[k].rank = k;
    }

    G2->arcnum += G2->vernum - 1;//对 G‘进行初始化
    win = (int**)malloc(sizeof(int) * (G2->vernum + 5));
    for (k = 0; k < G2->vernum; k++)
        win[k] = (int*)malloc(sizeof(int) * (G2->vernum + 5));
    win2 = (int**)malloc(sizeof(int) * (G2->vernum + 5));
    for (k = 0; k < G2->vernum; k++)
        win2[k] = (int*)malloc(sizeof(int) * (G2->vernum + 5));
    for (k = 0; k < G2->vernum; k++)
        for (l = 0; l < G2->vernum; l++) {
            win[k][l] = MAX;
            win2[k][l] = MAX;
        }
    for (k = 0; k < G2->arcnum; k++)
        win2[G2->arc[k].i][G2->arc[k].j] = G2->arc[k].w;

    int* D = (int*)malloc(sizeof(int) * (G2->vernum * G2->vernum + 5));
    int* h = (int*)malloc(sizeof(int) * (G2->vernum + 5));
```

```
    if (!BellmanFord(G2, &G2->ver[s], win2)) {//调用 Bellman_ford 算法来判断是否存在负环
        printf("the input graph contains a negative-weight cycle.\n");
        return 0;
    }
    else {
        for (k = 0; k < G2->vernum; k++) {
            h[k] = G2->ver[k].d;
        }
        for (k = 0; k < G2->arcnum; k++) {
            win[G2->arc[k].i][G2->arc[k].j] = win2[G2->arc[k].i][G2->arc[k].j] + h[G2->arc[k].i] - h[G2->arc[k].j];//对边权重新赋值
        }
        for (k = 0; k < G->vernum * G->vernum; k++)
            *(D + k) = 0;
        for (k = 0; k < G->vernum; k++) {
            Dijkstra(G, &G->ver[k], win);//调用 Dijkstra 算法来求每个顶点的单源最短路径
            printGraph(k, G, fp);
            for (l = 0; l < G->vernum; l++)
                *(D + k * G->vernum + l) = G->ver[l].d + h[l] - h[k];
        }
    }
    return D;
}
```

主函数

略

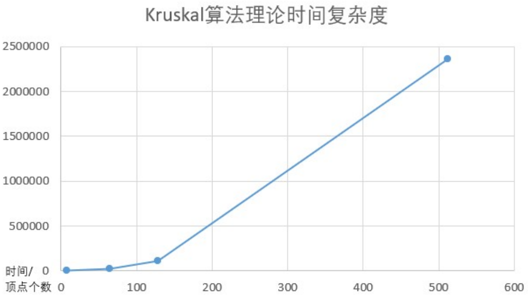
# 四、 实验结果与分析

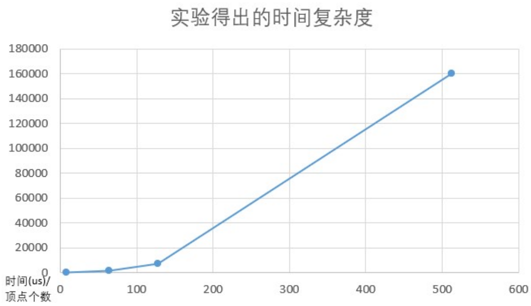
## 1. Kruskal 算法分析

输出结果（以较小数据规模为例）：

```
the whole cost is 40
2 1 2
2 3 2
4 6 2
3 7 5
2 0 6
0 4 10
7 5 13
```

理论时间复杂度和实际时间复杂度：



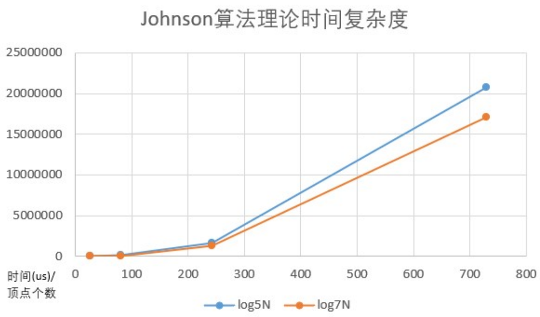

从结果可以看出，理论时间复杂度($N^2*lg(N)$)与实验得出的时间复杂度趋势较为符合。

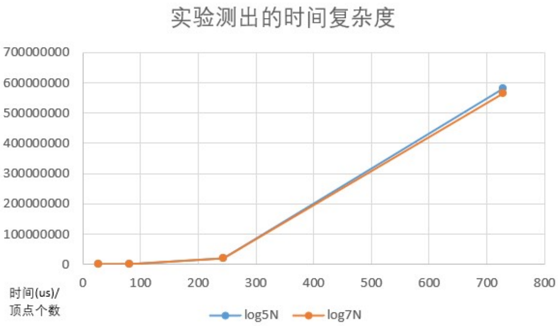
## 2. Johnson 算法分析

部分输出结果（以较小数据规模为例）：

```
(0, 1 no exist)
(0,14,16,13,2 22)
(0,14,16,13,3 29)
(0,14,4 31)
(0,14,16,13,3,5 66)
(0,14,16,13,3,6 45)
(0, 7 no exist)
(0,14,16,13,3,6,8 50)
(0,14,4,9 37)
(0, 10 no exist)
(0,14,4,9,25,11 39)
(0,14,16,22,19,18,24,12 69)
(0,14,16,13 22)
(0,14 5)
(0,14,4,9,25,11,21,15 52)
(0,14,16 18)
(0,14,16,13,3,6,8,17 84)
(0,14,16,22,19,18 45)
(0,14,16,22,19 37)
(0, 20 no exist)
(0,14,4,9,25,11,21 49)
(0,14,16,22 35)
(0,14,4,9,25,11,21,15,23 58)
```

理论时间复杂度和实际时间复杂度：




从结果可以看出，理论时间复杂度(EV*lg(N))与实验得出的时间复杂度趋势较为符合。而在两个不同出边规模下，实验中的两者差距不大，这可能是因为两者出边差别只有一条，所以区别并不是别特大，且本次实验中未来维护最短路径，还增加了很多维护路径的操作，这导致在每次执行完 Dijkstra 算法之后都要增加一些 O(VlgV)的操作，导致算法的时间复杂度发生了一些与 E 无关的变化，故会造成以上的实验结果。