

集合论实验报告：最短路径

一.实验背景

1.实验目的

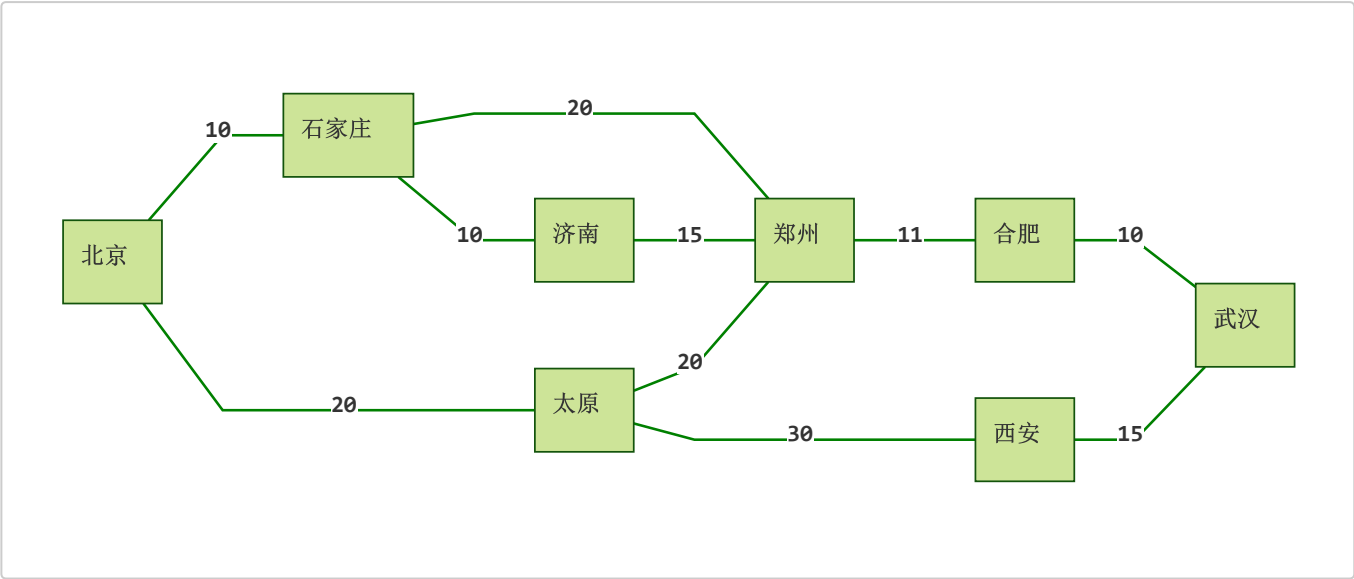
在实际生活中，我们常常会遇到需要这样的问题，比如：

- 一批货从北京到武汉的的最快，或最省钱的走法。
- 在城市群中建一个仓储基地，建在什么位置可以让各个城市的送货速度都比较快。

实验目标就是解决诸如上述的各类问题。

2.实验描述

诸如以上问题，我们都可以将其转化为图论中的最短路径问题。
将以上问题可以抽象为如下情形：



将各个位置考虑为图的顶点，而距离或者所用时间则考虑为图的边权。
则可以利用最短路径算法求解。
此处采用迪杰斯特拉算法。

二.实验原理

1.迪杰斯特拉算法思想

- 设 $G=(V,E)$ 是一个带权有向图，把图中顶点集合 V 分成两组：

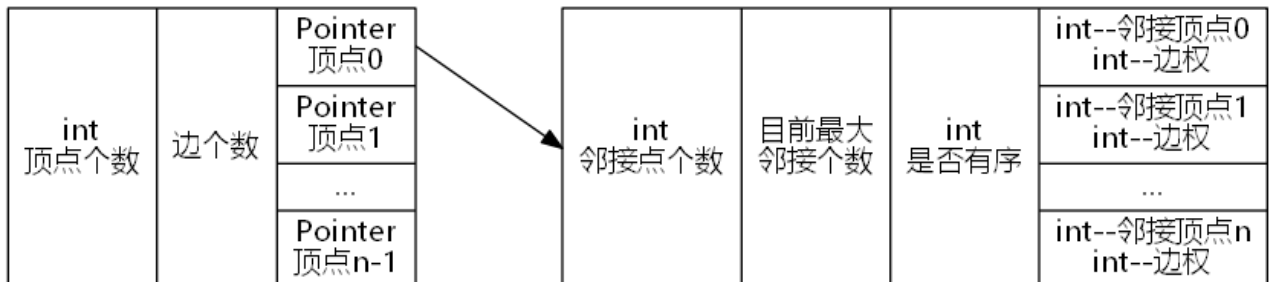
1. 第一组为已求出最短路径的顶点集合（用S表示）
 2. 第二组为其余未确定最短路径的顶点集合（用U表示）
- 初始时S中只有一个源点，以后每求得一条最短路径，就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了。
 - 按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。
 - 此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

2.迪杰斯特拉算法步骤

1. 初始时，S只包含源点，即 $S = \{v\}$ ，v的距离为0。U包含除v外的其他顶点，即： $U = \{\text{其余顶点}\}$ ，若v与U中顶点u有边，则正常有权值，若u不是v的出边邻接点，则权值为 ∞ 。
2. 从U中选取一个距离v最小的顶点k，把k，加入S中（该选定的距离就是v到k的最短路径长度）。
3. 以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值，修改后的距离值的顶点k的距离加上边上的权。
4. 重复步骤2和3直到所有顶点都包含在S中。

三·代码实现

1. 设计数据结构



参照了耶鲁大学的一位前辈的代码，使用可变数组（利用malloc和realloc），既有了数组的快速，又有了可改变数组大小的能力，在空间效率和时间效率上都有极好的效果。为了代码的简洁，在这里去掉了错误检测。

```
//无向带权图数据结构
typedef struct _list {
    int vec;           //邻接顶点
    int weight;        //权
} link_list;

typedef struct w_graph {
    int n;             //顶点个数
    int m;             //边个数
    struct successors {
        int d;         //临接点个数
        int len;       //最大临接点个数
        char is_sorted; //
        link_list list[1]; //临接列表
    };
};
```

```
    } *v_list[1];  
} *WGraph;
```

2. 设计数据操作

代码实现如下:

创建一个顶点从0 ~ n-1的带权图

```
WGraph w_graph_create(int n) {  
    WGraph g;  
    int i;  
    g = malloc(sizeof(struct w_graph) + sizeof(struct successors *) * (n - 1));  
    g->n = n;  
    g->m = 0;  
    for (i = 0; i != n; i++) {  
        g->v_list[i] = malloc(sizeof(struct successors));  
        g->v_list[i]->d = 0;  
        g->v_list[i]->len = 1;  
        g->v_list[i]->is_sorted = 1;  
    }  
    return g;  
}
```

释放内存

```
void w_graph_destroy(WGraph g) {  
    int i;  
    for (i = 0; i != g->n; i++) {  
        free(g->v_list[i]);  
    };  
    free(g);  
}
```

添加边和权

```
void w_graph_add_edge(WGraph g, int u, int v, int weight) {  
    //是否需要增长list  
    while (g->v_list[u]->d >= g->v_list[u]->len) {  
        g->v_list[u]->len *= 2;  
        g->v_list[u] =  
            realloc(g->v_list[u], sizeof(struct successors) + sizeof(link_list) * (  
    }  
    //添加新临接点  
    g->v_list[u]->list[g->v_list[u]->d].vec = v;  
    g->v_list[u]->list[g->v_list[u]->d].weight = weight;  
    g->v_list[u]->d++;  
    g->v_list[u]->is_sorted = 0;  
    //边数+1  
    g->m++;  
}
```

~~~C

返回顶点个数

```
int w_graph_vector_count(WGraph g) {  
    return g->n;  
}
```

返回边个数

```
int w_graph_edge_count(WGraph g) {  
    return g->m;  
}
```

返回顶点的度

```
int w_graph_out_degree(WGraph g, int source) {  
    return g->v_list[source]->d;  
}
```

判断是否邻接

```
int w_graph_has_edge(WGraph g, int source, int sink) {  
    int i;  
    if (w_graph_out_degree(g, source) >= BSEARCH_THRESHOLD) {  
        //确保已经被排序  
        if (!g->v_list[source]->is_sorted) {  
            qsort(g->v_list[source]->list,  
                  g->v_list[source]->d,  
                  sizeof(link_list),  
                  list_cmp);  
        }  
        //使用二分查找  
        link_list to_find;  
        to_find.vec = sink;  
        to_find.weight = 0;  
  
        return bsearch(&to_find,  
                       g->v_list[source]->list,  
                       g->v_list[source]->d,  
                       sizeof(link_list),  
                       list_cmp) != 0;  
    } else {  
        //数据量很少，直接遍历  
        int vec_degree = g->v_list[source]->d;  
        for (i = 0; i != vec_degree; i++) {  
            if (g->v_list[source]->list[i].vec == sink) {  
                return 1;  
            }  
        }  
    }  
}
```

```

}
return 0;
}

```

获取边的权,和判断是否有边的接口思想一致,不在赘述

```
int w_graph_weight_edge(WGraph g, int source, int sink);
```

提供读取顶点数据的接口

```

void w_graph_foreach(WGraph g,int source,
    void (*f)(WGraph, int, int, int, void *),void *data){
    int i;
    for (i = 0; i != g->v_list[source]->d; ++i) {
        f(g, source, g->v_list[source]->list[i].vec,
            g->v_list[source]->list[i].weight, data);
    }
}

```

### 3. 实现算法

- 定义所需数据结构

```

struct min_len {
    int n;                //顶点个数
    int vec;              //起始点
    struct list {
        int dist;         //与起始顶点的距离
        int prev;         //前驱动点
    } a_list[1];          //
};

```

- 实现算法

```

Min_len Dijkstra(WGraph g, int source) {
    int i, j, *S;
    Min_len res;
    int vec_num = w_graph_vector_count(g);
    res = malloc(sizeof(struct min_len) + sizeof(struct list) * (vec_num - 1));
    S = calloc((size_t) vec_num, sizeof(int));
    res->n = vec_num;
    res->vec = source;
    //初始化各点距离及标记
    for (i = 0; i != vec_num; ++i) {
        S[i] = 0;
        if (w_graph_has_edge(g, source, i)) {
            res->a_list[i].dist = w_graph_weight_edge(g, source, i);
            res->a_list[i].prev = source;
        } else {

```

```

        res->a_list[i].prev = -1;
        res->a_list[i].dist = INFINITY;
    }
}
//初始点已知
res->a_list[source].dist = 0;
res->a_list[source].prev = source;
S[source] = 1;

for (i = 1; i != vec_num; ++i) {
    int min_dst = INFINITY;
    int u = source;
    //找出未使用过的点中dist最小的
    for (j = 0; j != vec_num; ++j) {
        if ((!S[j]) && res->a_list[j].dist < min_dst) {
            u = j; //u是距离source最近的点
            min_dst = res->a_list[j].dist;
        }
    }

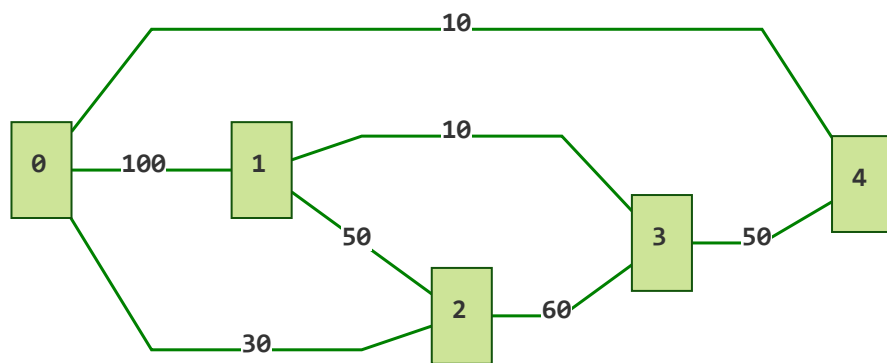
    S[u] = 1; //将u标记为已使用

    for (j = 0; j != vec_num; ++j)
        if ((!S[j]) && w_graph_has_edge(g, u, j)) { //j点未被使用且u,j之间有
            if (res->a_list[u].dist + w_graph_weight_edge(g, u, j) < res->a_list[j].dist)
                res->a_list[j].dist = res->a_list[u].dist + w_graph_weight_edge(g,
                res->a_list[j].prev = u;
            }
        }
    }
}
free(S);
return res;
}

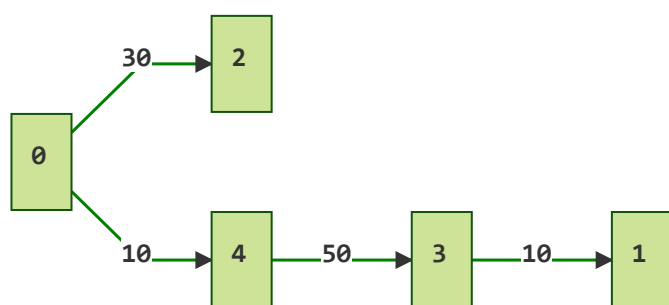
```

## 四·实验结果

输入下图：



可得到下列结果：



## 五·算法分析

1. 迪杰斯特拉算法的时间复杂度是 $O(n^2)$ ,空间复杂度则取决于数据结构，使用矩阵则为 $O(n^2)$ 。
2. 相对于弗洛伊德算法迪杰斯特拉算法来说，迪杰斯特拉算法不能处理边权为负值的情况。
3. 迪杰斯特拉算法通过斐波那契堆优化后的复杂度为 $O(E + V \lg(V))$ ，显著提高效率，限于时间原因，此处并未优化。