7.图论

算法模板来自AcWing
最短路宫水三叶讲解

0 概念

- 曼哈顿距离：只能上下左右四个方向走的最短距离

1 图的存储

- 有向图add1次，无向图add2次
- 稀疏图(n m)用邻接表
- 稠密图(n m^2)用邻接矩阵
- 存在重边取最短的边

1-1 邻接矩阵

1-2 邻接表

- 数组实现

```
int N;
vector<int> dot(N);
vector<vector<int>> g;
bool vis[N] = false;

void add(int a, int b){
    edges[a].push_back(b);
}
```

- 链表实现

```
//用数组模拟单链表
int N;
int head[N]; //每个点的链表头指向的位置
int val[N]; //链表对应的值
int next[N]; //每个点的next值
int idx; //用到数组的第几个位置了
bool vis[N] = false;


//初始化
idx = 0;
memset(head, -1, sizeof head;

//添加边a->b
void add(int a, int b){
    val[idx] = b; //存邻接表里的终点
    next[idx] = h[a]; //当前边指向原来的最先边
```

```
    head[a] = idx++; //链表头指向当前边
}
```

2 搜索

2-1 深度优先搜索

邻接矩阵

```
void dfs(int dot){
    vis[dot] = true;

    for(int i = 0; i < g[i].size(); i++){
        int nextDot = g[dot][i]
        if(!vis[nextDot]) dfs(nextDot);
    }
}
```

邻接表

```
int dfs(int d){
    vis[d] = true;

    //h[u]代表点i邻接表指向的第一个节点
    for(int i = head[d]; i != -1; i = next[i]){
        int nextDot = val[i];
        if(!vis[nextDot]) dfs(nextDot);
    }
}
```

2-2 广度优先搜索

邻接矩阵

```
void bfs(){
    queue<int> q;
    vis[0] = true;
    q.push(dot[0]);

    while(!q.empty()){
        int d = q.front();
        q.pop();

        for(int i = 0; i < g[d].size(); i++){
            int nextDot = g[d][i];
            if(!vis[nextDot]){
                vis[nextDot] = true;
                q.push(nextDot);
            }
        }
    }
```

```
}
```

邻接表

```
void bfs(){
    queue<int> q;
    vis[0] = true;
    q.push(1);

    while(!q.empty()){
        int d = q.front();
        q.pop();

        for(int i = head[d]; i != -1; i = next[i]){
            int j = val[i];
            if(!vis[j]){
                vis[j] = true;
                q.push(j);
            }
        }
    }
}
```

题目

- 863.二叉树中所有距离为 K 的结点

2-3 A*启发式搜索

3 拓扑排序 关键路径(到所有终点用时最长的路径)

- 题目:
- 207.课程表
- 210. 课程表 II
- 2050. 并行课程 III（拓扑排序+关键路径）
- 851. 喧闹和富有

理论

- 参考博客：知乎
- 拓扑序不唯一、只能在有向无环图中

邻接矩阵

```
int n;
vector<int> indegree; //入度
vector<vector<int>> edges;//储存边，仔细看懂含义，第二维存的是终点
void Topological Order(){
    for(int i = 0; i < relations.size(); i++){
        int out = relations[i][0], in = relations[i][1];
```

```
        indegree[in]++;
        edge[out].push_back(in);
    }
        queue<int> q;
        vector<int>ans; //最终排序后的序列
        for(int i = 0; i < n; i++)
    {
        if(indegree[i] == 0) q.push(i);
    }
    while(!q.empty()){
            int x = q.front();
            q.pop();
            ans.push_back(x);
            for(int i = 0; i < edges[x].size(); i++){
                    indegree[edges[x][i]]--;
                    if(indegree[edges[x][i]] == 0) q.push(edges[x][i]);
                }
        }
    if(ans.size() != n) //说明无解
}
```

```
void sloution(){

}

```

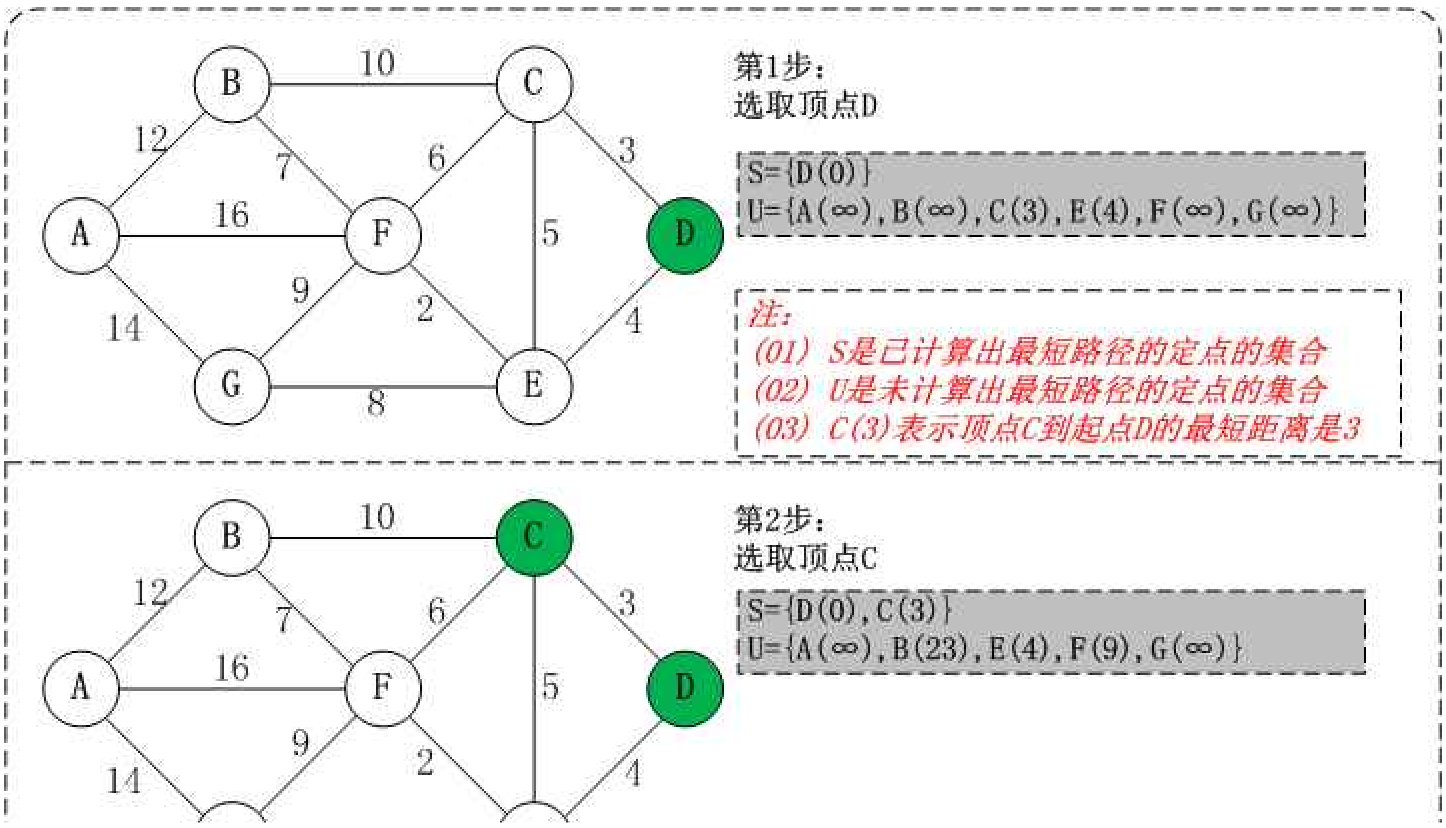
邻接表

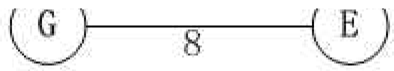
4 最短路

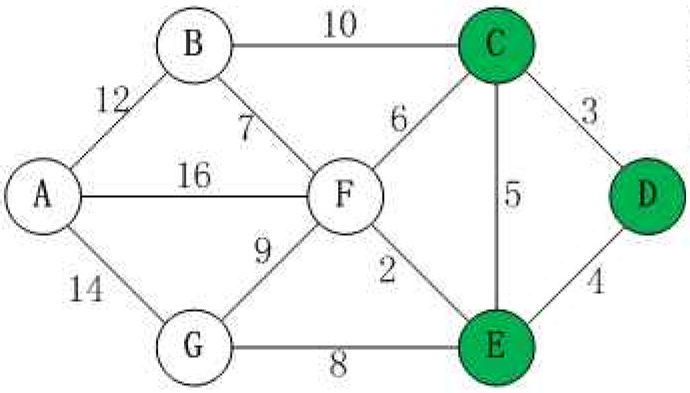
- 题目：743. 网络延迟时间

4-1 Dijkstra

- 边权都是正数才能用
- 图解链接

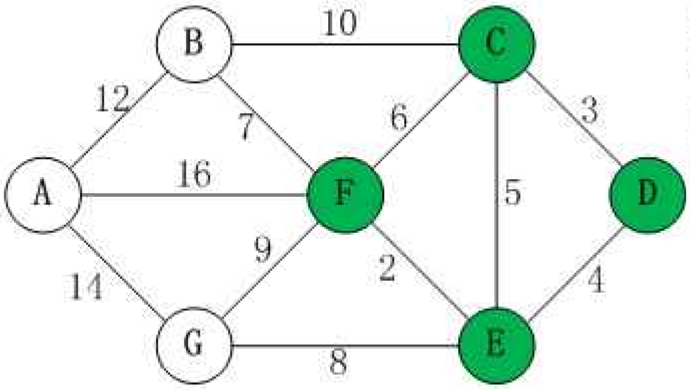






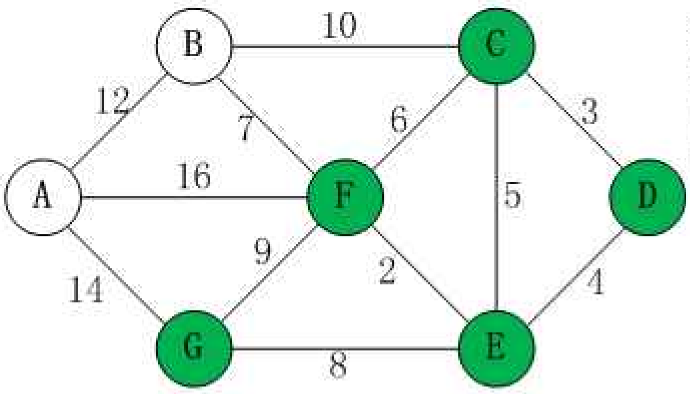
第3步：
选取顶点E

S={D(0),C(3),E(4)}
U={A(∞),B(23),F(6),G(12)}



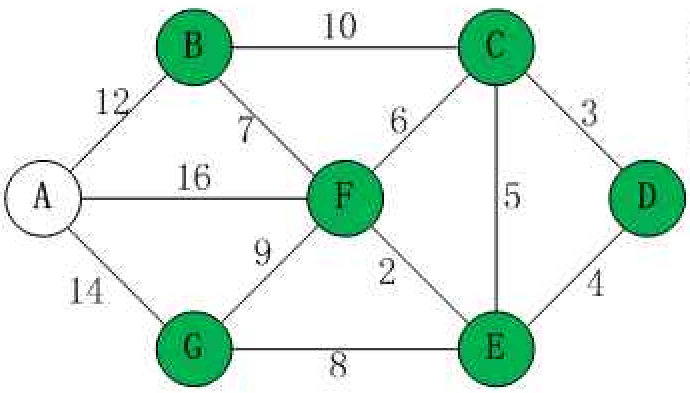
第4步：
选取顶点F

S={D(0),C(3),E(4),F(6)}
U={A(22),B(13),G(12)}



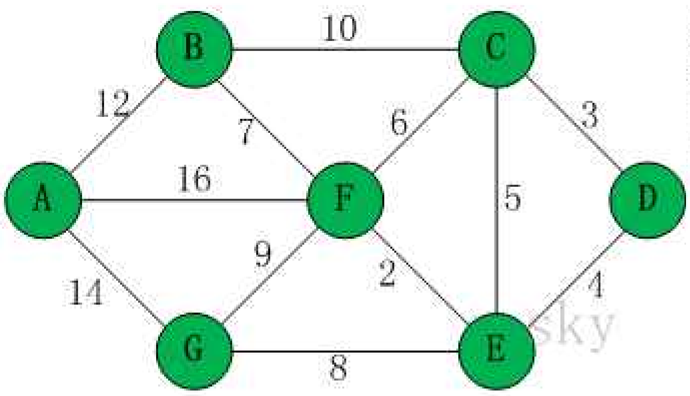
第5步：
选取顶点G

S={D(0),C(3),E(4),F(6),G(12)}
U={A(22),B(13)}



第6步：
选取顶点B

S={D(0),C(3),E(4),F(6),G(12),B(13)}
U={A(22)}



第7步：
选取顶点A

S={D(0),C(3),E(4),F(6),G(12),B(13),A(22)}

朴素Dijkstra （O(n^2)）

```
int N;
int g[N][N];
```

```
int dist[N]; //当前点到起点的距离
int uesd[N]; //当前点最短路已经确定

int Dijkstra(){
    memset(g, 0x3f, sizeof g);
    memset(dis, 0x3f, sizeof dis);
    for(){完成g邻接矩阵的填充}
    //used[k] = 1; //得用第一个点更新一遍最短路径，要不然都是max
    dis[1] = 0;

    //1已经加进去了，迭代n-1次
    for(int i = 1; i <= n; i++){
        int t = -1;

        //找最近点
        for(int j = 1; j <= n; j++){
            if(!used[j] && (t == -1 || dis[j] < dis[t])) t = j;
        }

        // if(t == n) break; // 优化：已经找到了最短路
        used[t] = true;

        //用最近点更新其他点
        for(int j = 1; j <= n; j++){
            dis[j] = min(dis[j], dis[t] + g[t][j]);
            //起点到t加t-j间距离；这个点到起点的距离
        }
    }

    if(dis[n] == 0x3f3f3f3f) return -1;
    return dis[n];
}
int main(){
    memset(g, 0x3f, sizeof g);
}
```

堆优化版dijkstra （mlogn）

```
typedef pair<int, int> PII;
int N;
int head[N], weight[N], val[N], next[N];//w是权重/边长
int dis[N]; //当前点到起点的距离
int vis[N]; //当前点最短路已经确定

void add(int a, int b, int c){
    //a是起点，b是终点，c是权重
    val[idx] = b;
    weight[idx] = c;
    next[idx] = h[a];
    heaf[a] = idx++;
}
int Dijkstra(){
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;

    //存的是距离、起点
    priority_queue<PII, vector<PII>, greater<PII> heap;
```

```
    heap.push({0, 1});

    while(!heap.empty()){
        //找到最近的点
        auto t = heap.top();
        heap.pop();

        int distance = t.first;
        int ver = t.second; //节点编号

        if(vis[ver]) continue;
        vis[ver] = true;

        ///用最近点更新其他点
        for(int i = head[ver]; i != -1; i = next[i]){
            int j = val[i];
            if(dis[j] > distance + w[i]){
                dis[j] =  distance + w[i];
                heap.push({dis[j], j});
            }
        }
    }
    if(dis[n] == 0x3f3f3f3f) return -1;
    return dis[n];
}
int main(){
    memset(head, -1, sizeof head);
}
```

- 2203. 得到要求路径的最小带权子图

4-2 Bellman-Ford算法（O(nm)）

- 如果有负权环，路径可能为-无穷
- 可以找负环，但是复杂度高

```
int n, m;
int dist[N];
int backup[N]; //做dist上次的备份，防止单次边循时候前面的数据影响了后面（dist[a] 影响了

struct Edge{
    int a, b, w;
}edges[M];

int bellman_ford(){

    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;
    //从头到尾最多不超过k条边
    for(int i = 0; i < k; i++){
        memcpy(backup, dist, sizeof dist);
        for(int j = 0; j < m; j++){
            int a = edge[j].a;
            int b = edge[j].b;
            int w = edge[j].w;
            dist[b] = min(dist[b], backup[a] + w);
        }
```

```
    }
    if(dist[n] > 0x3f3f3f3f / 2) return -1;
    return dist[n];
}
```

4-3 spfa 算法（O(mn) O(m)）

- 没有负环就可以用

```
int n;      // 总点数
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N];        // 存储每个点到1号点的最短距离
bool st[N];     // 存储每个点是否在队列中

void add(int a, int b, int c){
    //a是起点，b是终点，c是权重
    val[idx] = b;
    weight[idx] = c;
    next[idx] = h[a];
    heaf[a] = idx++;
}

int spfa()
{
    memset(dist, 0x3f, sizeof dist);
    dist[1] = 0;

    queue<int>q;
    q.push(1);
    st[1] = true;

    while(!q.empty()){
        int t = q.front();
        q.pop();

        s[t] = false;
        for(int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            if(dist[j] > dist[t] + w[i]){
                dist[j] = dist[t] + w[i];
                if(!st[j]){
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    if (dist[n] == 0x3f3f3f3f) return -1;
    return dist[n];
}
```

spfa判断图中是否存在负环

```
int n;      // 总点数
```

```
int h[N], w[N], e[N], ne[N], idx;       // 邻接表存储所有边
int dist[N], cnt[N];        // dist[x]存储1号点到x的最短距离，cnt[x]存储1到x的最短路
bool st[N];     // 存储每个点是否在队列中

// 如果存在负环，则返回true，否则返回false。
bool spfa()
{
    // 不需要初始化dist数组
    // 原理：如果某条最短路径上有n个点（除了自己），那么加上自己之后一共有n+1个点，由抽屉

    queue<int> q;
    //所有点全部放进队列
    for (int i = 1; i <= n; i ++ )
    {
        q.push(i);
        st[i] = true;
    }

    while (q.size())
    {
        auto t = q.front();
        q.pop();

        st[t] = false;

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[t] + w[i])
            {
                dist[j] = dist[t] + w[i];
                cnt[j] = cnt[t] + 1;
                if (cnt[j] >= n) return true;       // 如果从1号点到x的最短路中包含
                if (!st[j])
                {
                    q.push(j);
                    st[j] = true;
                }
            }
        }
    }

    return false;
}
```

4-4 floyd算法

```
初始化:
void init(){
    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;
}


// 算法结束后，d[a][b]表示a到b的最短距离
```

```
void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                //d[k, i, j] = d[k - 1, i, k] + d[k -1, k, j];
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

4-5 多源BFS

- 单源BFS只有一个起点，多源BFS有多个起点
- 剑指 Offer II 107. 矩阵中的距离

5 最小生成树

- 有无正负边没关系

5-1 prim

- 找最近的点

```
int n;      // n表示点数
int g[N][N];        // 邻接矩阵，存储所有边
int dist[N];        // 存储其他点到当前最小生成树的距离
bool st[N];     // 存储每个点是否已经在生成树中

const INF = 0X3f3f3f3f;

// 如果图不连通，则返回INF(值是0x3f3f3f3f), 否则返回最小生成树的树边权重之和
int prim()
{
    memset(dist, 0x3f, sizeof dist);

    int res = 0;
    for (int i = 0; i < n; i ++ )
    {
        int t = -1;
        for (int j = 1; j <= n; j ++ )
            if (!st[j] && (t == -1 || dist[j] < dist[t]))
                t = j;

        //说明图是不连通的
        if (i && dist[t] == INF) return INF;
        if (i) res += dist[t]; //不能再最后，循环会更新dist[t]
        st[t] = true;

        //这个点到集合的距离
        for (int j = 1; j <= n; j ++ ) dist[j] = min(dist[j], g[t][j]);
    }

    return res;
}
```

5-2 Kruskal算法

- 加最短的边

```
int n, m;       // n是点数，m是边数
int p[N];       // 并查集的父节点数组

struct Edge     // 存储边
{
    int a, b, w;

    bool operator< (const Edge &W)const
    {
        return w < W.w;
    }
}edges[M];

int find(int x)     // 并查集核心操作
{
    if (p[x] != x) p[x] = find(p[x]);
    return p[x];
}

int kruskal()
{
    sort(edges, edges + m);

    for (int i = 1; i <= n; i ++ ) p[i] = i;    // 初始化并查集

    int res = 0, cnt = 0;
    for (int i = 0; i < m; i ++ )
    {
        int a = edges[i].a, b = edges[i].b, w = edges[i].w;

        a = find(a), b = find(b);
        if (a != b)     // 如果两个连通块不连通，则将这两个连通块合并
        {
            p[a] = b;
            res += w;
            cnt ++ ;
        }
    }

    if (cnt < n - 1) return INF;
    return res;
}
```

6 二分图

6-1 染色法

```
vector<int> color; //0表示未染色，-1表示黑色，1表示白色

bool dfs(vector<vector<int>>& g, int x, int c){
    color[x] = c;
```

```
    for(int i = 0; i < g[x].size(); i++){
        int j = g[x][i];
        if(color[j] == 0){
            if(!dfs(g, j, -c)) return false;
        }
        else if(color[j] == c) return false;
    }
    return true;
}
bool isBipartite(vector<vector<int>>& graph) {
    int n = graph.size();
    color.resize(n);
    for(int i = 0; i < n; i++){
        if(color[i] == 0){
            if(!dfs(graph, i, 1)) return false;
        }
    }
    return true;
}
```

6-2 匈牙利算法

labuladong

- 797 图基础
- 207 210 拓扑
- 743 1514 1631 dijtesila
- 785 886 二分图
- 261 1135 1584 Kruskal

欧拉图

Hierholzer算法

```
    void Hierholzer(unordered_map<int, vector<int>>& edges, int dot){
        //得是&，在edges上操作，否则有死循环
        auto& v = edges[dot];
        while(v.size()){
            int nextdot = v.back();
            v.pop_back();
            Hierholzer(edges, nextdot);
        }
        //路径结果是倒序的
        path.push_back(dot);
    }
    vector<vector<int>> validArrangement(vector<vector<int>>& pairs) {
        unordered_map<int, vector<int>> edges;
        unordered_map<int, int> indegree;
        unordered_map<int, int> outdegree;
        for(auto&& x : pairs){
            outdegree[x[0]]++;
            indegree[x[1]]++;
            edges[x[0]].push_back(x[1]);
```

```
        }

        //找起点，要么入度 = 出度 + 1，要么所有点都能是起点
        int start = pairs[0][0];
        for(auto& x : edges) {
            int node = x.first;
            if(outdegree[node] ==  indegree[node] + 1) {
                start = node;
            }
        }

        //Hierholzer找路径
        Hierholzer(edges, start);
        vector<vector<int>> ans;
        //把倒序的路径反过来
        reverse(path.begin(), path.end());
        for(int i = 0; i < path.size() - 1; i++){
            ans.push_back({path[i], path[i + 1]});
        }

        return ans;
    }
```

- 题解
- 2097. 合法重新排列数对