网络流_2 `EK` 算法与`Dinic`算法

FORD-FULKERSON方法

维护一个残留网络，不断地迭代找增广路，将增广路删除后得到新的残留网络中继续迭代，直到某次迭代中找不到增广路时，此时当前流为最大流

```
FORD-FULKERSON(G,s,t)
for each edge(u,v) in G.E:
    (u,v).f=0
while there exits a path p from s to t in the residual network Gf: #当存在残留网络
    Cf(p)=min{Cf(u,v):(u,v) is in p} #找到增广路径的残存容量, 由定义为p上最小边
    for each edge (u,v) in p:
        if(u,v) in E:
            (u,v).f=(u,v).f+Cf(p)
        else:
            (u,v).f=(v,u).f-Cf(p)
```

`Edmonds-Karp`算法

```
while(进行迭代)
    1.从起点到终点BFS找增广路
    2.更新残留网络:
        (1)正向边减去k
        (2)反向边增加k
```

板子:

```
/*链式前向星存图的好处是可以快速找到反向边 edge^1*/
int n,m,S,T;
int h[N],e[M],f[M],ne[M],idx; //f[]存储残留网络容量
int q[N],d[N],pre[N]; //q[]为bfs用队列,d[]为增广路径的残存容量,pre[N]为记录的反向边
bool st[N];

void add(int a,int b,int c){
    e[idx] = b, f[idx] = c, ne[idx] = h[a] , h[a] = idx++;
    e[idx] = a, f[idx] = 0, ne[idx] = h[b] , h[b] = idx++;
}

bool bfs(){
    int hh = 0 , tt = 0;
    memset(st,0,sizeof st);
    q[0] = S, st[S] = true , d[S] = INF;
    while(hh <= tt){
        int t = q[hh ++];
        for(int i = h[t];~i;i=ne[i]){
            int ver = e[i];
            if(!st[ver] && f[i]){ //f[]存的边的容量, 应该大于0
                st[ver] = true;
                d[ver] = min(f[i],d[t]);
                pre[ver] = i;
                if(ver == T) return true;
                q[++ tt] = ver;
```

```
            }
        }
    }
    return false;
}

int EK(){
    int r = 0;
    while(bfs()){ //每次迭代找增广路
        r += d[T];
        for(int i = T;i != S;i = e[pre[i]^1])
            f[pre[i]] -= d[T],f[pre[i] ^ 1] += d[T];
    }
    return r;
}
```

Python

```
N = 100010
INF = 0x3f3f3f3f
# init n表示点数，m表示边数,S起点，T终点，idx内存分配器
n = 0; m = 0; S = 0; T = 0; idx = 0;
e = [ 0 for _ in range(2*N) ]; ne = [ 0 for _ in range(2*N) ]; h = [ -1 for _ in
range(N) ]; f = [0 for _ in range(N)] # graph
q = [0 for _ in range(N)]; d = [0 for _ in range(N)]; pre = [0 for _ in
range(N)]  # bfs用队列 增广路上残留容量 前驱结点
st = [False for _ in range(N)] # 状态

def add(a, b, c):
    e[idx] = b; f[idx] = c; ne[idx] = h[a] ; h[a] = idx ; idx += 1;
    e[idx] = a; f[idx] = c; ne[idx] = h[b] ; h[b] = idx; idx += 1;

def bfs():
    hh = 0 ; tt = 0
    for i in range(len(st)): # 初始化遍历数组
        st[i] = 0
    q[0] = S ; st[S] = True; d[S] = INF
    while( hh <= tt):
        t = q[hh]; hh += 1;
        i = h[t]
        while(~i):
            ver = e[i]
            if((not st[ver]) and f[i]):
                st[ver] = True
                d[ver] = min(f[i],d[i])
                pre[ver] = i
                if(ver == T):
                    return True
                t += 1
                q[t] = ver
            i = ne[i]
    return False

def EK():
    r = 0
    while(bfs()):
        r += d[T]
```

```
        i = T
        while(i != S):
            f[pre[i]] -= d[T]
            f[pre[i] ^ 1] += d[T]
            i = e[pre[i] ^ 1]
    return r
```

`DINIC`算法 O(n^2m)

优化思路：每次增广不止增广一条路，通过爆搜的方式将所有可以增广的路径。但是为了避免图中出现环导致死循环，因此引入`分层图`的概念

> `分层图的层数`：从起点开始`bfs`，第一次遍历到某点便是某个点对应的层数

然后用`dfs`统一搜索出来可以增广的路径，然后一起增广

`DINIC伪代码`：

```
while there exits a path p from s to t in the residual network :
    BFS(); //建立分层图
    while find a:
        ans += a
return ans
```

`板子`：

```
/*int dinic(){
    int ans = 0, flow = 0;
    while(bfs()) while(flow = find(S,INF)) ans += flow;
    return ans;
}*/
const int INF = 1e8;
int n,m,S,T;
int e[M],ne[M],h[N],f[M],idx=0;
int q[M],d[N],cur[N]; //层数，当前弧优化

void add(int a,int b,int c){
    e[idx] = b, f[idx] = c, ne[idx] = h[a] , h[a] = idx ++ ;
    e[idx] = a, f[idx] = 0, ne[idx] = h[b] , h[b] = idx ++ ;
}

bool bfs(){
    int hh = 0 , tt = 0;
    memset(d,-1,sizeof d);
    q[0] = S,d[S] = 0,cur[S] = h[S];

    while(hh <= tt){
        int u = q[ hh ++ ];
        for(int i=h[u];~i;i=ne[i]){
            int ver = e[i];
            if(d[ver] == -1 && f[i]){
                d[ver] = d[u] + 1;  //初始化分层图
                cur[ver] = h[ver];  //当前弧初始化为当前链式前向星的第一个边
                if(ver == T) return true;
                q[++ tt] = ver;
```

```
            }
        }
    }
    return false; //没有找到增广路径
}

int find(int u,int limit){
    if(u == T) return limit;
    int flow = 0; //准备从u点向后找最大的增广路上的残留容量

    for(int i = cur[u]; ~i && flow < limit;i = ne[i]){  //flow < limit 必加优化
        cur[u] = i;
        int ver = e[i];
        if(d[ver] == d[u] + 1 && f[i]){
            int t = find(ver,min(f[i],limit-flow));
            if(!t) d[ver] = -1; //如果此边到达不了终点，就直接删除
            f[i] -= t, f[i ^ 1] += t , flow += t;
        }
    }
    return flow;
}

int dinic(){
    int ans = 0, flow = 0;
    while(bfs()) while(flow = find(S,INF)) ans += flow;
    return ans;
}
```

`优化`：

- 当前弧优化：由于考虑到邻接表存边的性质，我们使用`cur[]`数组来存储当搜索到某个节点之后，接下来应该搜索哪条边。比如第一条边已经满了，则说明我们需要搜索邻接表顺序下的第二、三...条边。
-

> 至于时间复杂度，y总说`EK`算法大概能够处理点数+边数在1000~10000的网络，`Dinic`处理10000~100000的网络
>
> 此外还有`ISAP`算法，`EK`算法使用来求最小费用流，此外还有`HLPP`预留推进算法（最快的）