基础算法与数据结构（十） 最大流最小割

流网络的基本概念和性质

1. 定义：流网络G = (V, E)是一个有向图，图中每条边有一个非负的容量值c，不会有一条边是另一条边的反向边（即不存在u->v 和 v->u同时存在）。且存在一个源点和一个汇点。
2. 容量限制：对于所有的节点，有0≤f(u,v)≤c(u,v) 现在的流量不大于最大容量。
3. 流量守恒：对于除了源点和汇点的节点，流入的流量等于流出的流量。

类流网络的转换

根据之前的定义，可以知道有一些带权图并不符合流网络的定义，下面将对两种可以转换为流网络的有向图进行分析：

1. 有反向边的图：反向边与正向边的容量可以互相抵消
2. 有多个源点汇点的图：创建两个节点，作为超级源点和超级汇点。
3. 无向图：创建一个相同容量的反向边。

最大流问题

本部分问题可以在纸上画一些简单的图进行模拟，可以更好帮助理解。（没错就是我懒得画图了。。。）

最大流问题的目标

在一个流网络中，找到一个最大流量，这个流量可以通过这个流网络。

最大流问题的一些基本概念

约定，正向边是原有网络中就有的边，反向边为新增的边。

1. 残存网络：仍有空间对网络流进行调整的边组成。简单来说，就是在原来的基础上，为每条边新增了一条反向边的图。这条反向边的容量初始化的时候为0，然后随着网络中流的增加，原有的边的容量逐渐较少，这条新增的反向边的容量等于当前在这条边上的流的值。
   残存网络严格意义上来讲不属于流网络，但其满足所有流网络的性质。
   利用残存网络可以知道当前还可以增加多大的流。
2. 增广路：是残存网络中一条从源节点到汇点的简单路径。当前增广路上可以增加的流量最多为原有正向边上的容量。
3. 残存容量：当前增广路p上可以为每条边增加的最大值。也就是残存网络中正向边的容量。
4. 图的割：图G = (V,E) 的一个切割(S,T) 将结点集合V划分为S和T = V-S 两个集合。
5. 割的净流量：f(S,T) 可以定义为所有从S中的结点到T中结点的流量的和，减去T中结点到S中结点流量的和。
6. 割的容量：c(S,T)等于所有从S中的结点到T中结点的容量的和。

7. 最小割：所有割中，容量最小的一个。
8. 最大流最小割定理：
   设f为流网络G = (V, E)中的一个流，该流网络的源点为s，汇点为t，那么下面三个条件是等价的：
   1） f是G的一个最大流 2） 残存网络不在包括增广路径 3）|f| = c(S,T)
   隐含意义：当一个流为最大流时，其必定为最小切割的容量。
9. 最小割的意义：保证没有一条从s到t的路径，需要删去的边的最小容量。

Ford-Fulkerson方法

这个方法依赖于之前所讲的最大流最小割定理，有这个定理的保证，可以确保这个方法最后计算出来的流就是最大流。

这个定理的证明可以参考算法导论的相关部分（P414）

语言描述为：每一次迭代对流进行增加，知道残存网络中不再存在一个增广路径。

伪代码描述：

```
FORD-FULKERSON-METHOD(G,s,t)
1. initialize flow f to 0
2. while there exists an augmenting path p in the residual network G
3.    augment flow f along p
4. return f
```

Ford-Fulkerson算法

上面那个是方法，就是一个最大流的方法，因为这个方法有很多不一样时间复杂度的实现方式，这部分主要讲这个算法一个基本实现，这个实现就是通过就的DFS进行搜索，时间复杂度估算为O(F|E|) 其中F为最大流的值，在实际应用中属于比较快，有更快的先使用BFS再DFS的算法，也附在后，但不进行详细注解。

```
//利用邻接表存储图来实现的一个最大流算法的实现
struct edge{
  int to;
  int cap;
  int rev;  //反向边的编号
}

vector<edge> G[MAX_V];   //用向量模拟的邻接表
bool used[MAX_V];
```

```
void addEdge(int from,int to,int cap){
  //加入正向边，反向边的编号正好为未加反向边是那个点所含有的边数
  G[from].push_back((edge){to, cap, G[to].size()});
  //加入反向边，反向边对应的反向边就是正向边，就是from的size减掉1
  G[to].push_back((edge){from,0,G[from].size()-1});
}

int dfs(int v,int t,int f){
  //到达结束点，返回当前的最大流
  if(v == t) return f;
  used[v] = true;
  for(int i = 0;i<G[v].size();i++){
      //遍历v的所有边
      egde &e = G[v][i];
      //to节点没有访问过，代表这条边不是返回的边，并且还有容量可以增加。
      if(!used[e.to] && e.cap > 0){
          //从to结点继续向前，最大流量就是现在的流量和残余流量中小的那个
          int d = dfs(e.to, t, min(f, e.cap);
          if(d > 0){
              e.cap -= d;
              G[e.to][e.rev].cap += d;
              return d;
          }
      }
  }
  //返回0意味着最后找到的d为0，也就是没有边可以从s->t了
  return 0;
}

int max_flow(int s,int t){
  int flow = 0;
  for(;;){
      memset(used,0,sizeof(used);
      int f = dfs(s, t, INF);
      //f返回0时就代表着在图中不再存在一个可以走通的增广路径，那么就是最大流了
      if(f == 0) return flow;
      //还有路径可以走，那就增加到最大流中。
      flow+=f;
```

```
    }
}
```

接下来介绍一个更快一点的最大流算法，这个算法的复杂度是O(|E||V|^2)
这个算法的思想就是利用一个广度优先搜索对所有的节点进行一个编号，这样就可以根据这个分层的编号简化计算最大流中的深度优先搜索。

代码如下：不做详细解释

```
struct edge{int to, cap, rev;}
vector<edge> G[max_v];
int level[max_v];   //顶点到源点的距离编号
int iter[max_v];    //当前弧，比当前弧小的已经无用了

void addEdge(int from,int to,int cap){
  //加入正向边，反向边的编号正好为未加反向边是那个点所含有的边数
  G[from].push_back((edge){to, cap, G[to].size()});
  //加入反向边，反向边对应的反向边就是正向边，就是from的size减掉1
  G[to].push_back((edge){from,0,G[from].size()-1});
}

void bfs(int s){
  memset(level, -1, sizeof(level));
  queue<int> que;
  level[s] = 0;
  que.push(s);
  while(que.empty()){
      int v = que.front();
      que.pop();
      for(int i = 0;i<G[v].size;i++){
          edge& e = G[v][i];
          if(e.cap > 0 && level[e.to] < 0){
              level[e.to] = level[v] + 1;
              que.push(e.to);
          }
      }
  }
}
```

```
void dfs(int v,int t,int f){
  //到达结束点，返回当前的最大流
  if(v == t) return f;
  for(int i = 0;i<G[v].size();i++){
      //遍历v的所有边
      egde &e = G[v][i];
      //继续向下一层，并且还有容量可以增加。
      if(e.cap > 0 && level[v] < level[e.to]){
          //从to结点继续向前，最大流量就是现在的流量和残余流量中小的那个
          int d = dfs(e.to, t, min(f, e.cap);
          if(d > 0){
              e.cap -= d;
              G[e.to][e.rev].cap += d;
              return d;
          }
      }
  }
  //返回0意味着最后找到的d为0，也就是没有边可以从s->t了
  return 0;
}

int max_flow(int s,int t){
  int flow = 0;
  for(;;){
      bfs();
      if(level[t] < 0) return flow;
      memset(iter,0,sizeof(iter));
      int f;
      while((f = dfs(s,t,INF)) > 0) {
          flow+=f;
      }
  }
}
```

最大流问题的一些内容就是这些，典型例题暂缺。
下一节将会讲述最大流问题的一个典型应用： 二分图匹配。