网络流基础篇——Edmond-Karp 算法

BY 纳米黑客

这是我的一个初学者教程系列的一部分，也是这个系列的第一篇文章，这个系列计划中将包括网络流，线段树，树状数组等一些初学者比较难以入门的内容。

因为是初学教程，所以我会尽量避免繁杂的数学公式和证明。也尽量给出了较为完整的代码。

本文的目标群体是网络流的初学者，尤其是看了各种 NB 的教程也没看懂怎么求最大流的小盆友们。本文的目的是，解释基本的网络流模型，最基础的最大流求法，即 bfs 找增广路法，也就是 EK 法，全名是 Edmond-Karp，其实我倒是觉得记一下算法的全名和来历可以不时的拿出来装一装。

比如说这个，EK 算法首先由俄罗斯科学家 Dinic 在 1970 年提出，没错，就是 dinic 算法的创始人，实际上他提出的也正是 dinic 算法，在 EK 的基础上加入了层次优化，这个我们以后再说，1972 年 Jack Edmonds 和 Richard Karp 发表了没有层次优化的 EK 算法。但实际上他们是比 1790 年更早的时候就独立弄出来了。

你看，研究一下历史也是很有趣的。

扯远了，首先来看一下基本的网络流最大流模型。

有 n 个点，有 m 条有向边，有一个点很特殊，只出不进，叫做源点，通常规定为 1 号点。另一个点也很特殊，只进不出，叫做汇点，通常规定为 n 号点。每条有向边上有两个量，容量和流量，从 i 到 j 的容量通常用 c[I,j]表示,流量则通常是 f[I,j]。通常可以把这些边想象成道路，流量就是这条道路的车流量，容量就是道路可承受的最大的车流量。很显然的，流量<=容量。而对于每个不是源点和汇点的点来说，可以类比的想象成没有存储功能的货物的中转站，所有“进入”他们的流量和等于所有从他本身“出去”的流量。

把源点比作工厂的话，问题就是求从工厂最大可以发出多少货物，是不至于超过道路的容量限制，也就是，最大流。

比如这个图。每条边旁边的数字表示它的容量。

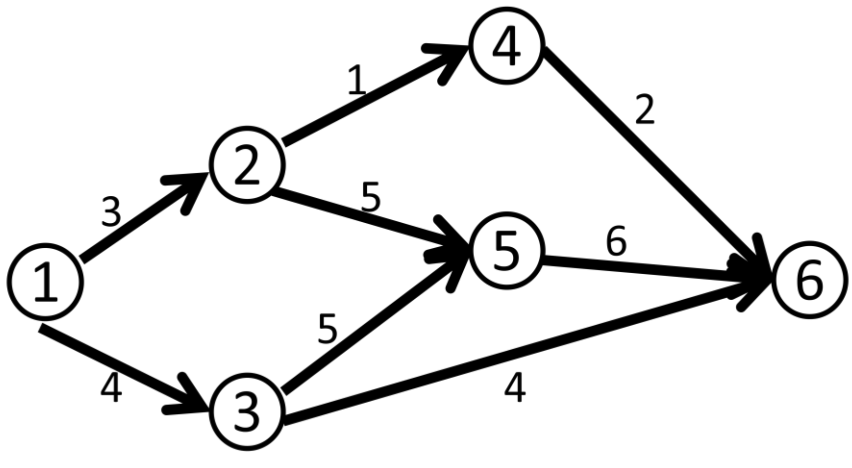

图 1

下面我们来考虑如何求最大流。

首先，假如所有边上的流量都没有超过容量(不大于容量)，那么就把这一组流量，或者说，这个流，称为一个可行流。一个最简单的例子就是，零流，即所有的流量都是 0 的流。我们就从这个零流开始考虑，假如有这么一条路，这条路从源点开始一直一段一段的连到了

汇点，并且，这条路上的每一段都满足流量<容量，注意，是严格的<,而不是<=。那么，我们一定能找到这条路上的每一段的(容量-流量)的值当中的最小值 delta。我们把这条路上每一段的流量都加上这个 delta，一定可以保证这个流依然是可行流，这是显然的。

这样我们就得到了一个更大的流，他的流量是之前的流量+delta，而这条路就叫做增广路。

我们不断地从起点开始寻找增广路，每次都对其进行增广，直到源点和汇点不连通，也就是找不到增广路为止。**当找不到增广路的时候，当前的流量就是最大流**，这个结论非常重要。

寻找增广路的时候我们可以简单的从源点开始做 bfs，并不断修改这条路上的 delta 量，直到找到源点或者找不到增广路。

这里要先补充一点，在程序实现的时候，我们通常只是用一个 c 数组来记录容量，而不记录流量，当流量+1 的时候，我们可以通过容量-1 来实现，以方便程序的实现。

Bfs 过程的半伪代码：

```
Procedure bfs;
  Begin
    Fillchar(data,data2,vis,pre){data 是队列数组，data2 是什么……自己研究下就明白了}
    Open=1;closed=0;delta=-1;
    Repeat
      Inc(closed);
      Inow=data[closed];
      If inow=n then
        Begin
          Delta=data2[closed]; Break
        End;
      For k=1 to n do
        If (c[Inow,k]>0)and(not vis[k]) then
          Begin
            Vis[k]=true
            Inc(open)
            Data[open]=k
            Data2[open]=min(data2[closed],c[Inow,k])
            Pre[k]:=Inow
          End;
    Until closed>=open
    If delta=-1 then exit
  {接下来修改容量}
    Y=n
    X=pre[n]
    Repeat
      Dec(c[x,y],delta);
      X=pre[x]
      Y=pre[y]
    Until y=1
```

```
  {修改最大流}
    Inc(tot,delta);
  End;
主过程部分：
While true do
  Begin
    Bfs;
    If delta=-1 then
      Begin
        writeln(tot);halt;
      end
  End
```

但事实上并没有这么简单，上面所说的增广路还不完整，比如说下面这个网络流模型。

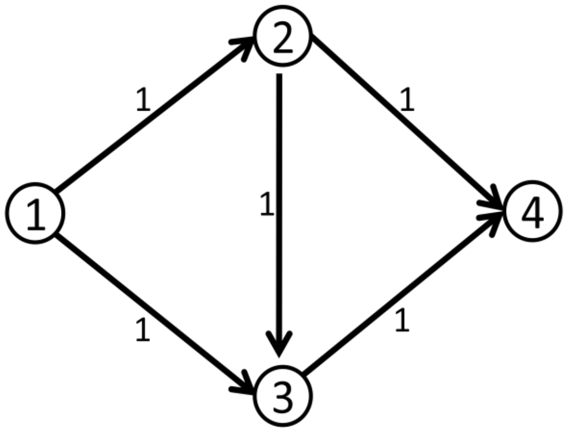

图 2

我们第一次找到了 1-2-3-4 这条增广路，这条路上的 delta 值显然是 1。于是我们修改后得到了下面这个流。（图中的数字是容量）

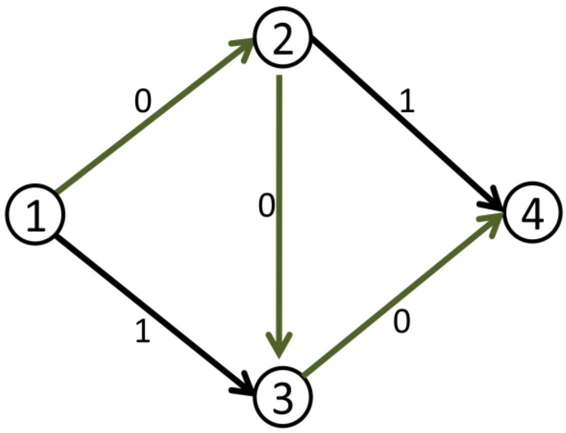

图 3

这时候(1,2)和(3,4)边上的流量都等于容量了，我们再也找不到其他的增广路了，当前的流量是 1。

但这个答案明显不是最大流，因为我们可以同时走 1-2-4 和 1-3-4，这样可以得到流量为 2 的流。

那么我们刚刚的算法问题在哪里呢？问题就在于我们没有给程序一个“后悔”的机会，应该有一个不走(2-3-4)而改走(2-4)的机制。那么如何解决这个问题呢？回溯搜索吗？那么我们的效率就上升到指数级了。

而这个算法神奇的利用了一个叫做反向边的概念来解决这个问题。即每条边(I,j)都有一条反向边(j,i)，反向边也同样有它的容量。

我们直接来看它是如何解决的：

在第一次找到增广路之后，在把路上每一段的容量减少 delta 的同时，也把每一段上的反方向的容量增加 delta。即在 Dec(c[x,y],delta)的同时，inc(c[y,x],delta)

我们来看刚才的例子，在找到 1-2-3-4 这条增广路之后，把容量修改成如下

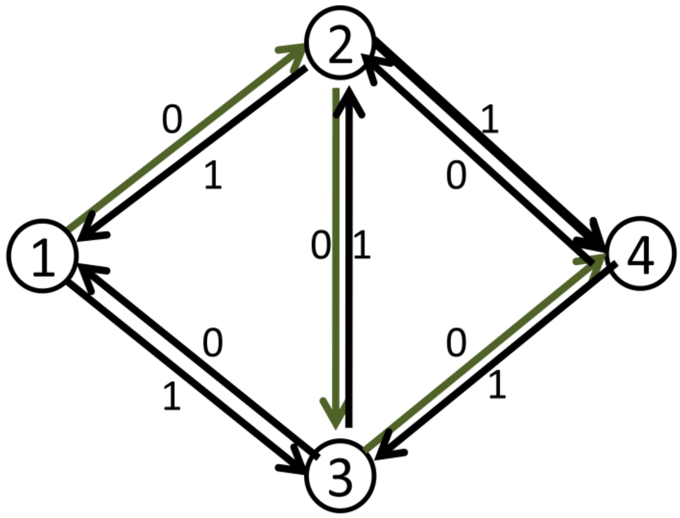

图 4

这时再找增广路的时候，就会找到 1-3-2-4 这条可增广量，即 delta 值为 1 的可增广路。将这条路增广之后，得到了最大流 2。

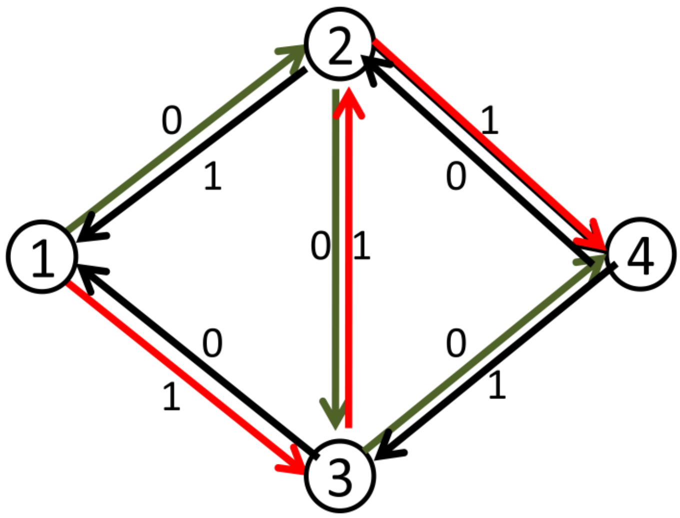

图 5

那么，这么做为什么会是对的呢？我来通俗的解释一下吧。

事实上，当我们第二次的增广路走 3-2 这条反向边的时候，就相当于把 2-3 这条正向边已经是用了的流量给“退”了回去，不走 2-3 这条路，而改走从 2 点出发的其他的路也就是 2-4。（有人问如果这里没有 2-4 怎么办，这时假如没有 2-4 这条路的话，最终这条增广路也不会存在，因为他根本不能走到汇点）同时本来在 3-4 上的流量由 1-3-4 这条路来“接管”。而最终 2-3 这条路正向流量 1，反向流量 1，等于没有流量。

这就是这个算法的精华部分，利用反向边，使程序有了一个后悔和改正的机会。而这个算法和我刚才给出的代码相比只多了一句话而已。

至此，最大流 Edmond-Karp 算法介绍完毕。接下来会介绍效率更高的 dinic 算法，最小割，最小费用最大流等内容。敬请期待。

本系列最新发布在 http://nano9th.wordpress.com.cn 和 http://www.namiheike.cn 上

{后面那个因为过期了我没续费暂时失效了，我会尽快修复……-_-}

最后说一句，如有疑问，建议，或者你想找男朋友，都请与我联系。

namiheike@gmail.com

QQ:411267858