从入门到精通: 最小费用流的“zkw算法”

by **zkw**, Jul 7, 2011, 12:48 am

1. 网络流的一些基本概念 点击阅读

很多同学建立过网络流模型做题目, 也学过了各种算法, 但是对于基本的概念反而说不清楚. 虽然不同的模型在具体叫法上可能不相同, 但是不同叫法对应的思想是一致的. 下面的讨论力求规范, 个别地方可能需要对通常的叫法加以澄清.

求解可行流: 给定一个网络流图, 初始时每个节点不一定平衡 (每个节点可以有盈余或不足), 每条边的流量可以有上下界, 每条边的当前流量可以不满足上下界约束. 可行流求解中没有源和汇的概念, 算法的目的是寻找一个可以使所有节点都能平衡, 所有边都能满足流量约束的方案, 同时可能附加有最小费用的条件 (最小费用可行流).

求解最大流: 给定一个网络流图, 其中有两个特殊的节点称为源和汇. 除源和汇之外, 给定的每个节点一定平衡. 源可以产生无限大的流量, 汇可以吸收无限大的流量. 标准的最大流模型, 初始解一定是可行的 (例如, 所有边流量均为零), 因此边上不能有下界. 算法的目的是寻找一个从源到汇流量最大的方案, 同时不破坏可行约束, 并可能附加有最小费用的条件 (最小费用最大流).

扩展的最大流: 在有上下界或有节点盈余的网络流图中求解最大流. 实际上包括两部分, 先是消除下界, 消除盈余, 可能还需要消除不满足最优条件的流量 (最小费用流), 找到一个可行流, 再进一步得到最大流. 因此这里我们的转化似乎是从最大流转化为可行流再变回最大流, 但其实质是将一个过程 (扩展的最大流) 变为了两个过程 (可行流 + 最大流).

以上概念同时适用于最小费用流.

2. 最小费用流的各种转化 点击阅读

时光真疯狂, 我一路执迷于匆忙.

zkw

About Owner

- **Posts:** 25
- **Joined:** Jun 21, 2007

Blog Stats

- **Blog created:** Aug 15, 2007
- **Total entries:** 5
- **Total visits:** 89664
- **Total comments:** 1

Search Blog

Enter term

Search

不少同学认为消圈算法比增广路算法使用面广, 可以有负边, 负圈, 每个点都能有盈余亏空等等. 实际上增广路算法也一样可以有负边, 负圈, 上下界等等, 且一般来说速度快于消圈算法.

以下讨论中 e 为盈余量, c 为费用, u 为容量.

1.最小费用(可行)流 $\rightarrow$ 最小费用最大流

建立超级源 s' 和超级汇 t' , 对顶点 i, 若 $e_i > 0$ 添加边 $s' \rightarrow i, c = 0, u = e_i$, 若 $e_i < 0$ 添加边 $i \rightarrow t', c = 0, u = -e_i$, 之后求从 s' 到 t' 的最小费用最大流, 如果流量等于 $\sum e_i$, 就存在可行流, 残量网络已在原图上求出.

2.最小费用最大流 $\rightarrow$ 最小费用(可行)流

连边 $t \rightarrow s, c = -\infty, u = +\infty$, 所有点 i 有 $e_i = 0$, 然后直接求.

3.最小费用(可行)流中负权边的消除

直接将负权边满流.

4.最小费用最大流中负权边的消除

先连边 $t \rightarrow s, c = 0, u = +\infty$, 使用 (3.) 中的方法消除负权边, 使用 (1.) 中的方法求出最小费用 (可行) 流, 之后**距离标号不变**, 再求最小费用最大流; 注意此时增广费用不能机械使用源点的标号——应该是源点汇点标号之差.

3. 费用流中的负边和负圈 点击阅读

在费用流的求解中难免会遇到负边和负圈的问题. 对于消圈算法, 负圈就是算法本身的一部分; 但对于增广路算法, 负圈是我们所不愿意看到的. 鉴于竞赛中使用消圈算法将带来严重的效率问题, 我们必须探索使用增广路算法处理费用流负边和负圈的方法.

对于单纯的负边, 如果这些负边没有组成负圈, 可以使用重赋权技术来处理, 即通过对每个节点合适的顶标, 使得 Reduced Cost 非负. 这个顶标通常可以使用到汇点的距离, 修改之后的边权变为 $c_{ij}^{\pi} = c_{ij} - D_i + D_j$. 根据流量平衡条件, 容易根据这个新的费用算出原费用, 同时可以证明这样的重赋权不改变最优解. 这样做的代价是一次 SPFA 操作, 时间耗费是相对较小的.

如果存在负圈, SPFA 算法将不会终止. 很多同学使用人为的限制条件使其终止, 这是错误的. 举一个例子: 源点和汇点均为孤立点, 图中另外存在一个负费用, 正容量的圈. 不管怎样初始标号, 仅凭增广无法消除这个负圈, 从而得不到最优解 (根据最小费用最大流的定义, 要在所有最大流 (流量都为 0) 中寻找一个费用最小的方案). 也许你会说这个例子太过极端, 但是也很容易构造出其他一些例子, 说明不处理负圈, 或者简单地对算法打补丁并不能解决本质问题.

解决方法就是将所有负费用的边强制满流, 称为“退流”操作. 这样做之后我们破坏了平衡条件, 但满足了最优条件. 之后我们可以先求可行流, 再求最大流, 其间一直维持最优条件, 并逐步解决平衡条件, 最大流条件等问题. 这也就是 (2.) 中提到的方法. 这个方法可以消除所有负权边 (在 Reduced Cost 意义下), 同时正确处理所有负圈问题. 那么, 这个方法的速度如何呢?

费用流相关的图大致可以分为两类, 一类图侧重于费用, 即总的流量不大, 如 Two Shortest (只有 2), KaKa's Matrix Travels 等. 较为通用的强制满流方法在这类图上表现不佳, 原因是原本的流量较小而负费用较多, 经过转化的图在可行流求解阶段流量与原流量相比很大, 严重影响速度. 另一方面, 这类图往往有深刻的最短路背景, 从而不会出现负圈, 可以使用 SPFA 重赋权. 另一类图侧重于流量, 即边的费用相差不大, 如 employee, 最优图像的求解. 这种图中流量不小而负费用相对有限 (也可以稍大, 关键是前者), 经过转化增加的流量可以很快完成计算, 因此强制满流方法就没有问题.

另外, 不同的建图方式有可能造成不同的模型. 在 employee 这道题中, 如果建图时从每个点向后连边, 从前向后运行最小费用最大流, 这时的算法就只有负边, 没有负圈. 而如果从每个点向前连边, 在整个图上求最小费用平衡流, 就会有负圈出现. 但是这里并没有本质区别: 将第二种模型中的所有向前连的边强制满流即可得到第一种模型, 如果将其他的负边也强制满流还能得到一个完全没有负边的模型. 大家可能也猜到了, 这几种模型在速度上相去不远, 毕竟根据我们刚才的讨论, 增加的流量在算法的执行中不占主要地位.

4. 最小费用流的“zkw算法” 点击阅读

最小费用流在 OI 竞赛中应当算是比较偏门的内容, 但是 NOI2008 中 employee 的突然出现确实让许多人包括 zkw 自己措手不及. 可怜的 zkw 当时想出了最小费用流模型, 可是他从来没有实现过, 所以不敢写, 此题 0 分. zkw 现在对费用流的心得是: 虽然理论上难, 但是写一个能 AC 题的费用流还算简单. 先贴一个我写的 costflow 程序: 只有不到 70 行, 费用流比最大流还好写 ~

点击查看程序代码

这里使用的是连续最短路算法. 最短路算法? 为什么程序里没有 SPFA? Dijkstra? 且慢, 先让我们回顾一下图论中最短路算法中的距离标号. 定义 D_i 为点 i 的距离标号, 任何一个最短路算法保证, 算法结束时对任意指向顶点 i 、从顶点 j 出发的边满足 $D_i \leq D_j + c_{ij}$ (条件1), 且对于每个 i 存在一个 j 使得等号成立 (条件2). 换句话说, 任何一个满足以上两个条件的算法都可以叫做最短路, 而不仅仅是 SPFA、Dijkstra, 算法结束后, 恰在最短路上的边满足 $D_i = D_j + c_{ij}$.

在最小费用流的计算中, 我们每次沿 $D_i = D_j + c_{ij}$ 的路径增广后都不会破坏条件 1, 但是可能破坏了条件 2. 不满足条件 2 的后果是什么呢? 使我们找不到每条边都满足 $D_i = D_j + c_{ij}$ 新的增广路. 只好每次增广后使用 Dijkstra, SPFA 等等算法重新计算新的满足条件 2 的距离标号. 这无疑是一种浪费. KM 算法中我们可以修改不断修改可行顶标, 不断扩大可行子图, 这里也同样, 我们可以在始终满足条件 1 的距离标号上不断修改, 直到可以继续增广 (满足条件 2).

回顾一下 KM 算法修改顶标的方法. 根据最后一次寻找交错路不成功的 DFS, 找到 $d = \min_{i \in V, j \notin V} \{-w_{ij} + A_i + B_j\}$, 左边的点增加 d, 右边的点减少 d. 这里也一样, 根据最后一次寻找增广路不成功的 DFS, 找到 $ d = \min_{i \in V, j \notin V, u_{ij} > 0} \left\{ c_{ij} - D_i + D_j \right\}$, 所有访问过的点距离标号增加 d. 可以证明, 这样不会破坏性质 1, 而且至少有一条新的边进入了 $D_i = D_j + c_{ij}$ 的子图.

算法的步骤就是初始标号设为 0 , 不断增广, 如果不能增广, 修改标号继续增广, 直到彻底不能

增广: 源点的标号已经被加到了 $+\infty$. 注意: 在程序中**所有的 cost 均表示的是 reduced cost**, 即 $c_{ij}^{\pi} = c_{ij} - D_i + D_j$. 另外, 这个算法不能**直接**用于有任何**负权边**的图. 更不能用于负权圈的情况. 有关这两种情况的处理, 参见 (2.) 和 (3.) 中的说明.

这样我们得到了一个简单的算法, 只需要增广, 改标号, 各自只有 7 行, 不需要 BFS, 队列, SPFA, 编程复杂度很低. 由于实际的增广都是沿最短路进行的, 所以理论时间复杂度与使用 SPFA 等等方法的连续最短路算法一致, 但节省了 SPFA 或者 Dijkstra 的运算时间. 实测发现这种算法常数很小, 速度较快, employee 这道题所有数据加在一起耗时都在 2s 之内.

5. “zkw” 费用流算法在哪些图上慢 点击阅读

实践中, 上面的这个算法非常奇怪. 在某一些图上, 算法速度非常快, 另一些图上却比纯 SPFA 增广的算法慢. 不少同学经过实测总结的结果是稠密图上比较快, 稀疏图上比较慢, 但也不尽然. 这里我从理论上分析一下, 究竟这个算法用于哪些图可以得到理想的效果.

先分析算法的增广流程. 和 SPFA 直接算法相比, 由于同属于沿最短路增广的算法, 实际进行的增流操作并没有太多的区别, 每次的增流路径也大同小异. 因此不考虑多路增广时, 增广次数应该基本相同. 运行时间上主要的差异应当在于如何寻找增流路径的部分.

那么 zkw 算法的优势在于哪里呢? 与 SPFA 相比, KM 的重标号方式明显在速度上占优, 每次只是一个对边的扫描操作而已. 而 SPFA 需要维护较为复杂的标号和队列操作, 同时为了修正标号, 需要不止一次地访问某些节点, 速度会慢不少. 另外, 在 zkw 算法中, 增广是多路进行的, 同时可能在一次重标号后进行多次增广. 这个特点可以在许多路径都费用相同的时候派上用场, 进一步减少了重标号的时间耗费.

下面想一想 zkw 算法的劣势, 也就是 KM 重标号方式存在的问题. KM 重标号的主要问题就是, 不保证经过一次重标号之后能够存在增广路. 最差情

况下, 一次只能在零权网络中增加一条边而已. 这时算法就会反复重标号, 反复尝试增广而次次不能增广, 陷入弄巧成拙的境地.

接下来要说什么, 大家可能已经猜到了. 对于最终流量较大, 而费用取值范围不大的图, 或者是增广路径比较短的图 (如二分图), zkw 算法都会比较快. 原因是充分发挥优势. 比如流多说明可以同一费用反复增广, 费用窄说明不用改太多距离标号就会有新增广路, 增广路径短可以显著改善最坏情况, 因为即使每次就只增加一条边也可以很快凑成最短路. 如果恰恰相反, 流量不大, 费用不小, 增广路还较长, 就不适合 zkw 算法了.

不适合怎么办? 继续向下看.

6. 最小费用流的原始对偶 (Primal-Dual) 算法 点击阅读

最小费用流的直接 SPFA 算法和前面说的 KM 重标号算法, 各自都有一些情况非常慢. 这里要写的就是一个所谓的“新算法” (其实非常经典), 融合两者优势.

先从 zkw 算法的本质开始. 细心的同学已经注意到了, 算法的主要过程就是反复交替进行最短路和最大流的计算. 具体地说, 每一次增广过程就是在零权网络中找一个最大流增广, 而重标号过程就是找到一组新的可行的距离标号. 既然如此, 最大流和最短路就完全可以用其他方法来替换. 这种交替进行最短路和最大流计算用来求得最小费用最大流的算法, 其实就是非常经典的原始对偶算法.

费用流的算法大致分为两种, 一种是经典的解法, 如消圈, 增广路, 原始对偶等等, 特点是步步为营, 维持可行性或者最优性其中之一, 再不断对另一方面作出改进. 另一种就比较现代一些, 典型的例子是松弛算法和网络单纯形, 由于放松了对求解过程中解的限制条件, 使得其速度远远超过经典解法, 同时也增加了编程难度和理解障碍. 下面要说的原始对偶算法, 速度自然不可能比松弛和网络单纯形快, 但应该是经典解法中的佼佼者了.

先考虑最短路算法的选择. 在流大, 费用小, 距离短的图上, KM 重标号就可以取得不错的效果, 但

是 SPFA 在另外的图上就很有优势.

再考虑最大流算法的选择. 最朴素的 SPFA 暴力增广中根本就不维护距离标号, 这里隐含地使用了一个单路增广. 在 zkw 费用流的算法里, 这里使用的是多路增广. 是不是有必要用更高级的网络流算法呢? 经过实际测试, 除非在特别稠密的图上, 否则效果一般. 原因是可增广的流量并不是很大, 如果去维护 SAP 的顶标反而增加了不少开销, 预流推进亦然.

总结一下, 折中的选择就是使用 Small Label First 优化 的 SPFA 来维护 zkw 算法中的距离标号, 保留多路增广. 注意, 除了第一次之外, SPFA 算法都工作在一个所有边均为正权的图上 (Reduced cost 意义下), 这是和不维护顶标直接 SPFA 的不同之处, 也是原来 zkw 算法的精神所在. 也正因为如此, 这里的 SPFA 甚至可以用 Dijkstra 替换.

这个特殊的原始对偶算法在稠密二分费用小的图上不敌原来的 zkw 算法, 但远远胜过暴力 SPFA. 在另外的图上, 对两者都是稳胜. 比暴力 SPFA 快原因是, 多路增广, 同时使用了 Reduced Cost 缩小了费用范围, 从而利于 SPFA 算法的工作 (需要的松弛次数减少), 而且使用 Reduced Cost 后不再有负边, 使 SLF 的优化落到了实处 (回忆: SLF 优化只有当所有边均为正的时候才能发挥出最佳效果), 甚至允许用 Dijkstra 来完成后面的工作. 比 zkw 算法快的原因是, 在流小费用大距离长的图上, 一次性把距离标号改对往往比反复调整更有效率.

最后是代码, 这次的例子是 POJ 3680 zkw Accepted 216K 360MS. 另外还有一个 POJ 3422, 是原来的 zkw 算法不擅长的一个例子, 这种方法只用 47MS, 代码大同小异, 就不放在这里了.

点击查看程序代码

```
#include <cstdio>
#include <cstring>
#include <deque>
#include <algorithm>
using namespace std;

const int V=440, E=V*2,
```

```
maxint=0x3F3F3F3F;

struct etype
{
    int t, c, u;
    etype *next, *pair;
    etype() {}
    etype(int T, int C, int U,
etype* N): t(T), c(C), u(U),
next(N) {}
    void* operator new(unsigned,
void* p){return p;}
} *e[V], Te[E+E], *Pe;

int S, T, n, piS, cost;
bool v[V];

void addedge(int s, int t, int c,
int u)
{
    e[s] = new(Pe++) etype(t, +c,
u, e[s]);
    e[t] = new(Pe++) etype(s, -c,
0, e[t]);
    e[s]->pair = e[t];
    e[t]->pair = e[s];
}

int aug(int no, int m)
{
    if (no == T) return cost += piS
* m, m;
    v[no] = true;
    int l = m;
    for (etype *i = e[no]; i; i =
i->next)
        if (i->u && !i->c && !v[i-
>t])
        {
            int d = aug(i->t, l <
i->u ? l : i->u);
            i->u -= d, i->pair->u
+= d, l -= d;
            if (!l) return m;
        }
```

```
    return m - l;
}

bool modlabel()
{
    static int d[V]; memset(d,
0x3F, sizeof(d)); d[T] = 0;
    static deque<int> Q;
Q.push_back(T);
    while(Q.size())
    {
        int dt, no = Q.front();
Q.pop_front();
        for(etype *i = e[no]; i; i
= i->next)
            if(i->pair->u && (dt =
d[no] - i->c) < d[i->t])
                (d[i->t] = dt) <=
d[Q.size() ? Q.front() : 0]
                    ?
Q.push_front(i->t) : Q.push_back(i-
>t);
    }
    for(int i = 0; i < n; ++i)
        for(etype *j = e[i]; j; j =
j->next)
            j->c += d[j->t] - d[i];
    piS += d[S];
    return d[S] < maxint;
}

int ab[V], *pab[V], w[V];

struct lt
{
    bool operator()(int* p1,int*
p2) {return *p1 < *p2;}
};

int main()
{
    int t;
    scanf("%d",&t);
    while(t--)
    {
```

```
        memset(e,0,sizeof(e));
        Pe = Te;
        static int m, k;
        scanf("%d %d", &m, &k);
        int abz = 0;
        for(int i = 0; i < m; ++i)
        {
            scanf("%d", pab[abz] =
&ab[abz]), abz++;
            scanf("%d", pab[abz] =
&ab[abz]), abz++;
            scanf("%d", &w[i]);
        }
        sort(&pab[0], &pab[abz],
lt());
        int c=0xDEADBEEF; n=0;
        for(int i = 0; i < abz;
++i)
        {
            if(c != *pab[i]) c =
*pab[i], ++n;
            *pab[i] = n;
        }
        ++n, S = 0, T = n++;
        for(int i = 0; i < T; ++i)
addedge(i, i+1, 0, k);
        for(int i = 0; i < m; ++i)
addedge(ab[i+i], ab[i+i+1], -w[i],
1);
        piS = cost = 0;
        while(modlabel())
            do memset(v, 0,
sizeof(v));
            while(aug(S, maxint));
        printf("%d\n", -cost);
    }
    return 0;
}
```

This post has been edited 7 times. Last edited by zkw, Jul 7, 2011, 2:16 am

No Comments Comment

Comment

0 Comments