最小费用最大流问题

前置知识

关于网络最大流, 这是本博客需要用到的两个算法: 网络最大流·初步

另外还有效率更高, 可是最小费用最大流不会用到的两个算法: 网络最大流·进阶

前置知识就只有第一篇博客的内容, 需要提前学习完再来看费用流.

简介

最小费用最大流问题, 简称费用流问题, Wikipedia 中的名称是: "Minimum-cost flow problem", 缩写为 "MCFP".

问题给出一个网络, 每条边除了有容量属性外, 还有一个属性: 单位流量费用. 也就是说, 一条边的费用是流经这条边的流量和这条边单位流量费用的乘积. 要求给出保证最大流的情况下的最小费用.

EK·改

EK 算法就是每次用 BFS 在残量网络上找到一条可以增广的路并且增广它. 为了防止反复横跳, 我们使用 BFS 而不是 DFS 寻找增广路, 单条增广路寻找复杂度 .

所以保证最大流, 不需要考虑增广路是怎么找的, 只要找到就可以, 所以可以使用最短路算法, 每次以 为起点, 以单位花费为边权, 每次求 到 的最短路, 这个最短路长度乘上这条增广路的流量计入答案, 然后进行下一轮的增广.

算法选择方面, 因为一条边的回边的边权是它的边权的相反数, 所以残量网络存在负权, 对于有负权的图的最短路, 我们使用 SPFA 算法.

Dinic·改

Dinic 一次求多条增广路, 而 SPFA 也是一次求 到所有点的最短路, 所以两者也可以结合.

具体操作是每次 BFS 分层的过程改为求 的单源最短路的过程. 在 DFS 时的规则也有所改变, 传统的 Dinic 中, 我们只允许流从一个点, 流向深度比它大 的点, 是为了保证流不会反复横跳 (我们设定的一切

规则都是为了规范流的方向而避免反复横跳, 从算法方面印证了规则之于自由的重要性). 在改进版算法中, 我们使用最短路来规范流的流动, 只要保证一条边是 到这条边的终点的最短路的一部分, 这条路就能走.

这个规则既保证了我们的选择一定是最短路, 又保证了流的有序性, 而且不破坏 Dinic 的特性: 多路增广.

实现起来也可以说比较写意, 但是需要注意的是, DFS 的过程和一般的 Dinic 相比增加了一个 标记.

这是因为最短路为序相比深度为序来说, 边权对顺序是有影响的, 也就是说当一条边和它的回边都有流量的时候, 它们构成一个 零环 , 意思是流量在这两点之间无论走几个来回, 花费都不变.

为了避免这种情况, 我们像 SPFA 一样使用 标记, 防止增广路径上一个点出现两次.

```
unsigned m, n, Hd, Tl;
int C, D, Flow(0), Cost(0), Tmp(0);
struct Node;
struct Edge {
  Node *To;
  Edge *Nxt;
  int Value, Contain;
}E[100005], *CntE(E - 1);
struct Node {
  Edge *Fst;
  int Dist;
  char InQue;
}N[5005], *S, *T, *A, *B, *Q[5005];
char SPFA() {
  Tl = Hd = 0;
  for (register unsigned i(1); i <= n; ++i) N[i].Dist = 0x3f3f3f3f;
  Q[++Tl] = S;
  S->Dist = 0;
  S->InQue = 1;
  register Node *x;
  while (Hd ^ Tl) {
    ++Hd; if(Hd > 5000) Hd -= 5000;
    x = Q[Hd];
    x->InQue = 0;
    register Edge *Sid(x->Fst);
    while (Sid) {
      if(Sid->Contain && Sid->To->Dist > x->Dist + Sid->Value) {
        Sid->To->Dist = x->Dist + Sid->Value;
        if(!(Sid->To->InQue)) {
          ++Tl; if(Tl > 5000) Tl -= 5000;
          Q[Tl] = Sid->To;
          Sid->To->InQue = 1;
        }
      }
      Sid = Sid->Nxt;
    }
  }
  return (T->Dist < 0x3f3f3f3f);
}
int DFS(Node *x, int Come){
  if(x == T) return Come;
  x->InQue = 1;
  register Edge *Sid(x->Fst);
  register unsigned Go, Flew(0);
  while (Sid) {
    if(Sid->To->Dist == x->Dist + Sid->Value && Sid->Contain && (!(Sid->To->InQue))) {
      Go = DFS(Sid->To, min(Sid->Contain, Come));
      Sid->Contain -= Go;
      Come -= Go;
      E[(Sid - E) ^ 1].Contain += Go;
```

```
      Cost += Sid->Value * Go;
      Flew += Go;
    }
    if(!Come) break;
    Sid = Sid->Nxt;
  }
  x->InQue = 0;
  return Flew;
};
int main() {
  n = RD(), m = RD(), S = N + RD(), T = N + RD();
  for (register unsigned i(1); i <= m; ++i) {
    A = N + RD(), B = N + RD(), C = RDsg(), D = RDsg();
    (++CntE)->Nxt = A->Fst;
    A->Fst = CntE;
    CntE->Contain = C;
    CntE->Value = D;
    CntE->To = B;
    (++CntE)->Nxt = B->Fst;
    B->Fst = CntE;
    CntE->Value = -D;
    CntE->To = A;
  }
  while (SPFA()) Flow += DFS(S, 0x7fffffff);
  printf("%d %d\n", Flow, Cost);
  return Wild_Donkey;
}
```

改·改

实际上叫做 "Primal-Dual Algorithm" (原始对偶算法).

就像 59·改 和 59 关系不大一样, 这个算法虽然名字原始, 但是我无论如何也不明白它和对偶有什么关系, 如果让我为他取名, 我觉得应该叫做:

由于 SPFA 的最坏复杂度可以达到 , 所以相当于在原算法的基础上增加了一个因子 , 可以说毫无效率可言.

所以考虑使用 Dijkstra, 但是图中有负权, Dijkstra 很难得到正确结果.

算法效率低是因为运行了多次 SPFA, 但是如果我们只运行一次 SPFA, 算法效率也是不受影响的.

一开始跑一遍 SPFA, 将 到每个点的最短路求出, 记为 , 我们称 为点 的 势 .

每次跑最短路, 发现因为残量网络越来越 残 , 到其它点的最短路只能变得更长, 不能变得更短.

每次跑 Dijkstra 之前, 假设我们已经求出了每个点的势, 也就是这个残量网络变得这么残之前的最短路, 那么规定每条边 的权值 .

因为上一次 之间的最短路一定不会比 更长, 否则最短路就变成 了, 所以容易知道 , 整理得 , 这样, 边权大于 , 我们就能使用 Dijkstra 了.

一条路径的权值和就变成了它们原来的权值之和加上起点的势再减去终点的势, 其意义是这条路径的起点终点之间的最短路在上次增广之后增长了多少. 也就是说, 我们用 Dijkstra 求的是一个增长量. ~~是最短路的导数~~

所以从新的权值求得的最短路求真正的最短路的方法也变得明了了, 就是用一开始 SPFA 求的最短路加上每一次增广后 Dijkstra 求的最短路的总和, 就是当前真正的最短路.

这样再拉去跑 Dinic/EK, 就和上面两个算法一样了, 但是网上的方法貌似都是用 EK, 但是理论上我们有了最短路, 就能像前面 `Dinic·改` 一样多路增广.

于是我就把 Dinic + Dijkstra + SPFA 写出来了, 很遗憾, 它获得了比上面的程序更低的效率.

```
unsigned m, n, Hd, Tl;
int C, D, Flow(0), Cost(0), Tmp(0);
struct Node;
struct Edge {
  Node *To;
  Edge *Nxt;
  int Vnew, Value, Contain;
}E[100005], *CntE(E - 1);
struct Node {
  Edge *Fst;
  int Dist, h;
  char InQue;
}N[5005], *S, *T, *A, *B, *Q[5005];
struct Que {
  Node *P;
  inline const char operator<(const Que &x) const{
    return this->P->Dist > x.P->Dist;
  }
}QTmp;
priority_queue <Que> Qu;
void SPFA() {
  Tl = Hd = 0;
  for (register unsigned i(1); i <= n; ++i) N[i].h = 0x3f3f3f3f;
  Q[++Tl] = S;
  S->h = 0;
  S->InQue = 1;
  register Node *x;
  while (Hd ^ Tl) {
    ++Hd; if(Hd > 5000) Hd -= 5000;
    x = Q[Hd];
    x->InQue = 0;
    register Edge *Sid(x->Fst);
    while (Sid) {
      if(Sid->Contain && Sid->To->h > x->h + Sid->Value) {
        Sid->To->h = x->h + Sid->Value;
        if(!(Sid->To->InQue)) {
          ++Tl; if(Tl > 5000) Tl -= 5000;
          Q[Tl] = Sid->To;
          Sid->To->InQue = 1;
        }
      }
      Sid = Sid->Nxt;
    }
  }
}
int DFS(Node *x, int Come){
  if(x == T) return Come;
  x->InQue = 1;
  register Edge *Sid(x->Fst);
  register unsigned Go, Flew(0);
```

```
  while (Sid) {
    if(Sid->To->h == x->h + Sid->Value && Sid->Contain && (!(Sid->To->InQue))) {
      Go = DFS(Sid->To, min(Sid->Contain, Come));
      Sid->Contain -= Go;
      Come -= Go;
      E[(Sid - E) ^ 1].Contain += Go;
      Cost += Sid->Value * Go;
      Flew += Go;
    }
    if(!Come) break;
    Sid = Sid->Nxt;
  }
  x->InQue = 0;
  return Flew;
};
char Dijkstra () {
  for (register unsigned i(1); i <= n; ++i) N[i].Dist = 0x3f3f3f3f;
  QTmp.P = S, Qu.push(QTmp);
  S->Dist = 0;
  S->InQue = 1;
  register Node *x;
  while (Qu.size()) {
    x = Qu.top().P;
    Qu.pop();
    x->InQue = 0;
    register Edge *Sid(x->Fst);
    while (Sid) {
      Sid->Vnew = Sid->Value + x->h - Sid->To->h;
      if(Sid->Contain && Sid->To->Dist > x->Dist + Sid->Vnew) {
        Sid->To->Dist = x->Dist + Sid->Vnew;
        if(!(Sid->To->InQue)) {
          QTmp.P = Sid->To;
          Qu.push(QTmp);
          Sid->To->InQue = 1;
        }
      }
      Sid = Sid->Nxt;
    }
  }
  for (register unsigned i(1); i <= n; ++i) N[i].h += N[i].Dist;
  return (T->h < 0x3f3f3f3f);
}
int main() {
  n = RD(), m = RD(), S = N + RD(), T = N + RD();
  for (register unsigned i(1); i <= m; ++i) {
    A = N + RD(), B = N + RD(), C = RDsg(), D = RDsg();
    (++CntE)->Nxt = A->Fst;
    A->Fst = CntE;
    CntE->Contain = C;
    CntE->Value = D;
    CntE->To = B;
```

```
    (++CntE)->Nxt = B->Fst;
    B->Fst = CntE;
    CntE->Value = -D;
    CntE->To = A;
  }
  SPFA();
  do {
    Flow += DFS(S, 0x7fffffff);
  } while(Dijkstra());
  printf("%d %d\n", Flow, Cost);
  return Wild_Donkey;
}
```