网络流算法

前言

- 图论这门古老而又年轻的学科在信息学竞赛中占据了相当大的比重。其中，网络流算法经常在题目中出现。网络流涵盖的知识非常丰富，从基本的最小割最大流定理到网络的许多变形再到最高标号预流推进的六个优化等等，同学们在平时需要多多涉猎这方面的知识，不断积累，才能应对题目的各种变化。
- 图论这门学科的诞生始于18世纪欧拉证明了七桥问题，发表《依据几何位置的解题方法》一文。但图论的真正发展是从20世纪五六十年代开始的。所以说，图论是一门既古老又年轻的学科。

基本概念

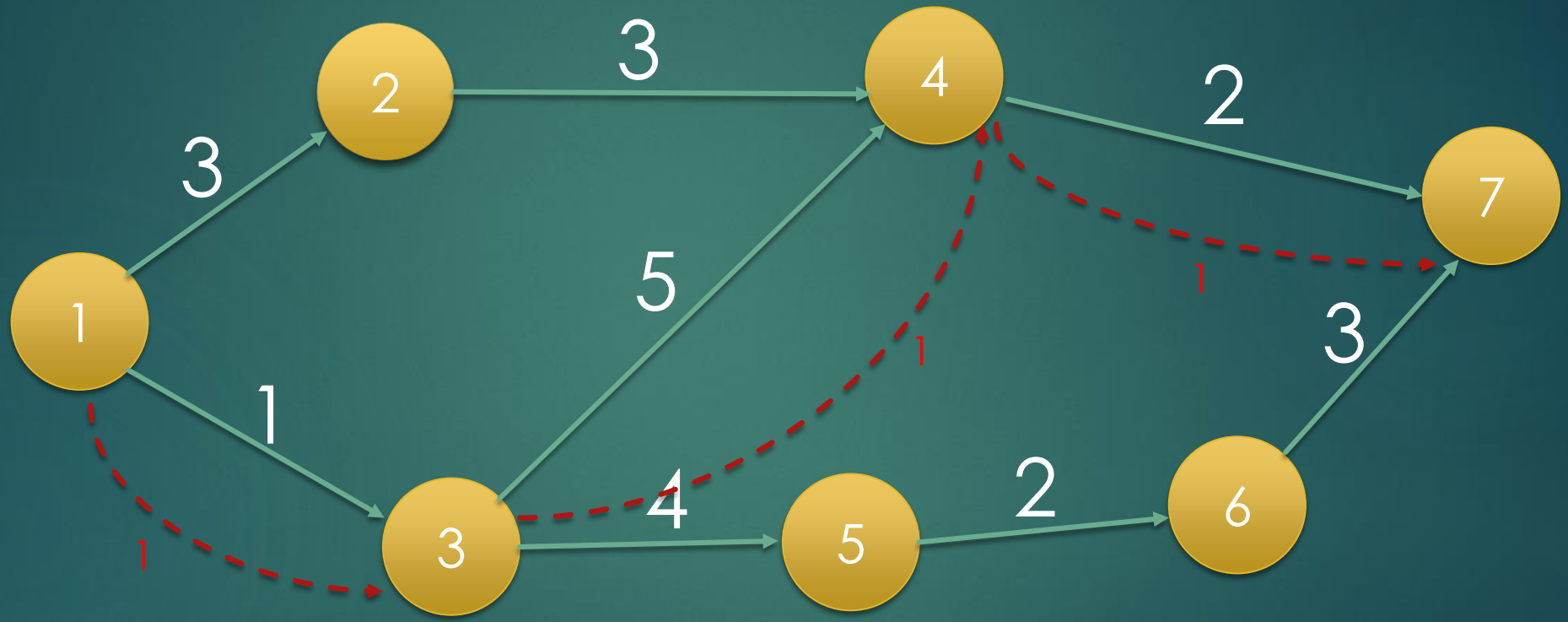
- 在有向图G（V，E）中，每条边（u,v）∈E上都有一个非负实数的容量c (u,v)。指定一个顶点s，称为源点，再指定另一个点t，称为汇点。
- 如果(u,v) ∉ E，我们假设c (u,v)=0

基本概念

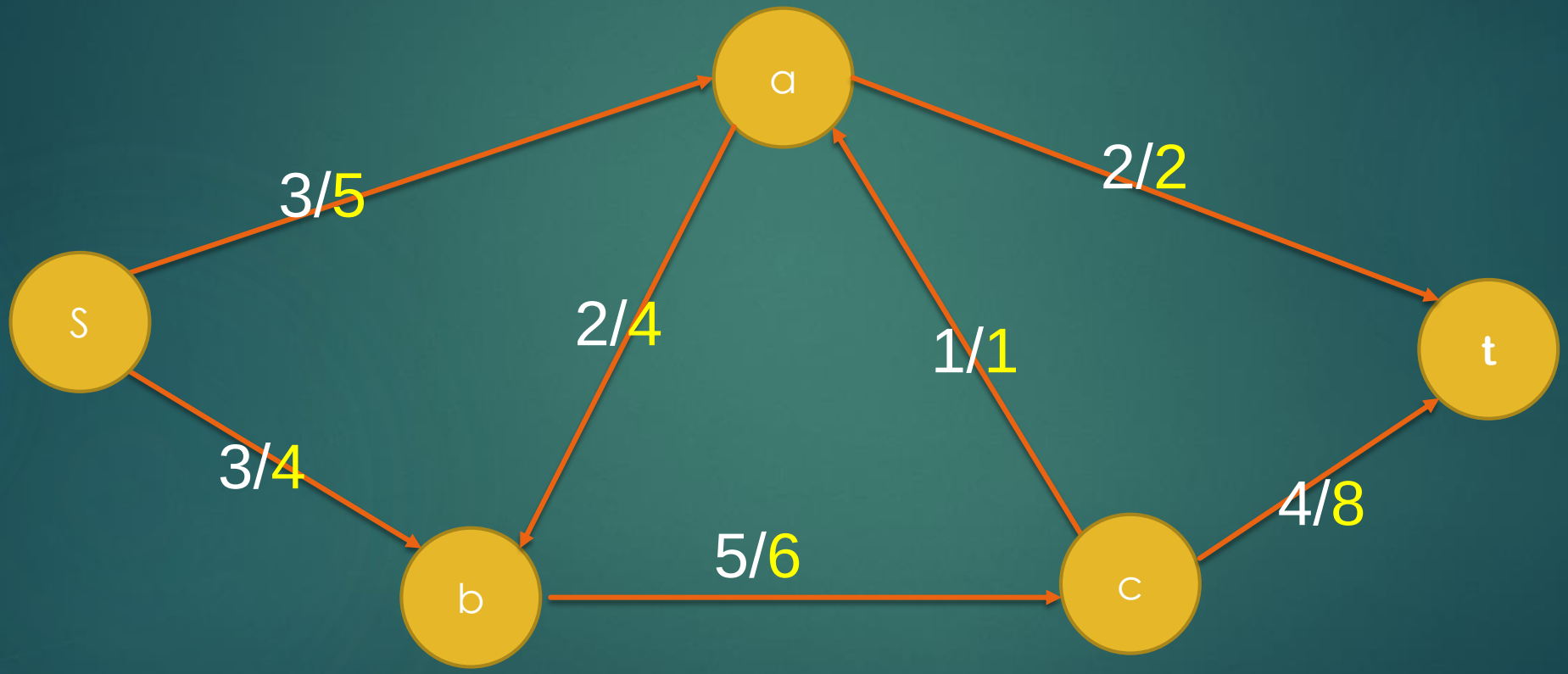
- 网络流是一个对于所有结点 v 都有以下特性的实数函数$f:V \times V \longrightarrow \Re$

1. 容量限制 (Capacity Constraints)
2. 斜对称 (Skew Symmetry)
3. 流守恒 (Flow Conservation)

例子



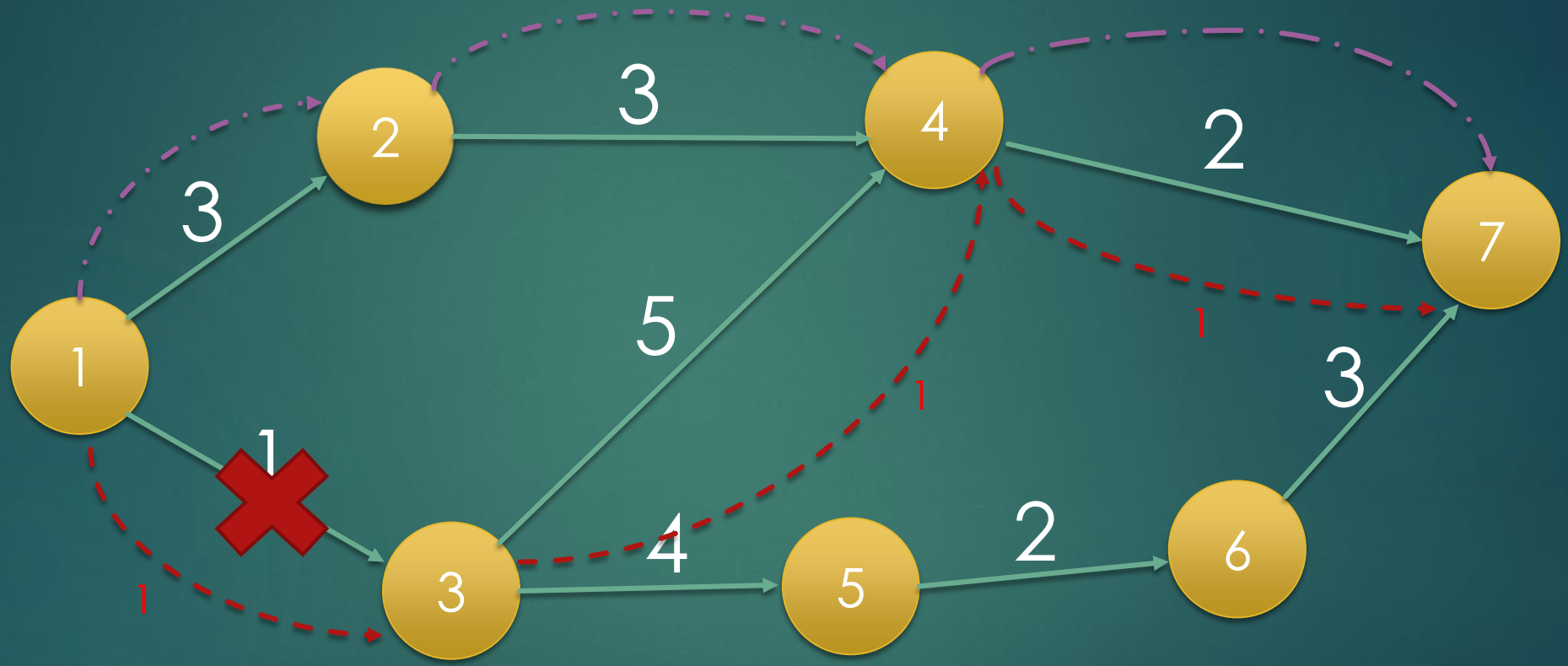
例子



考虑这样的贪心算法

1. 找一条S到t 的只经过f(e) < c(e)的边。
2. 如果不存在满足条件的路径，则算法结束。否则，沿着该路径尽可能的增加f(e)，返回第一步。

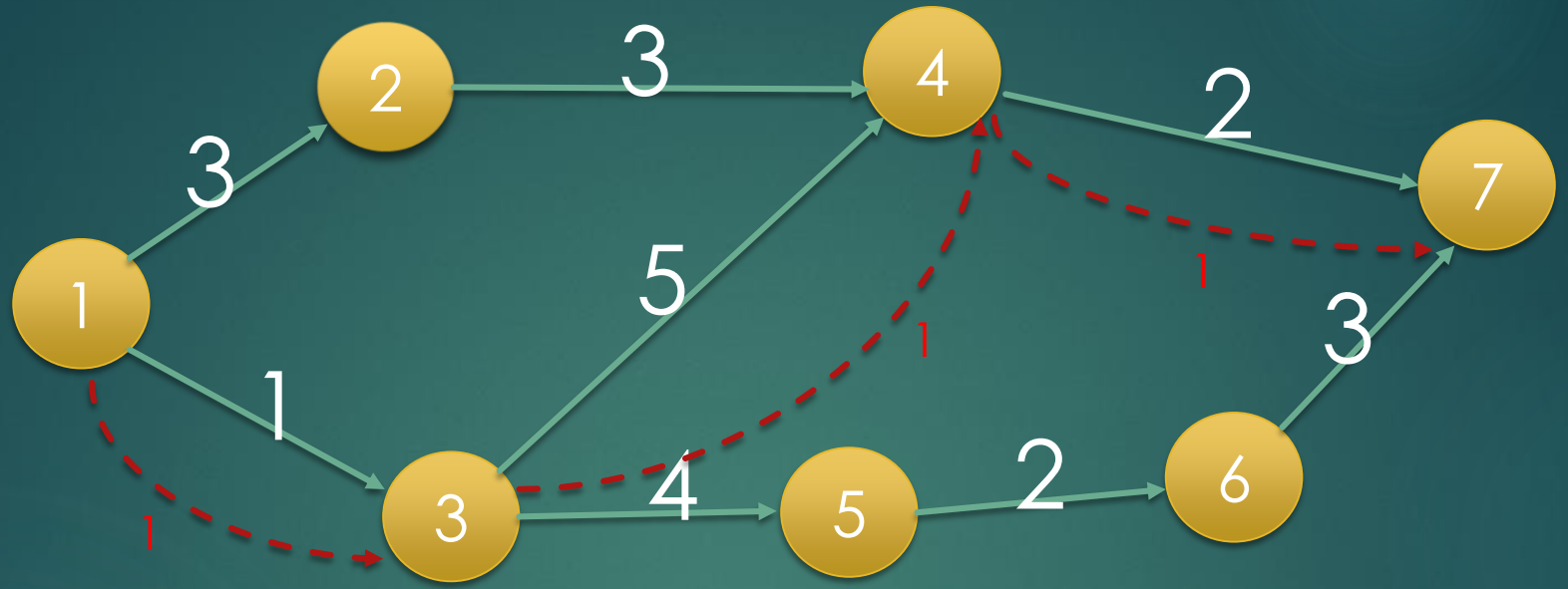
例子

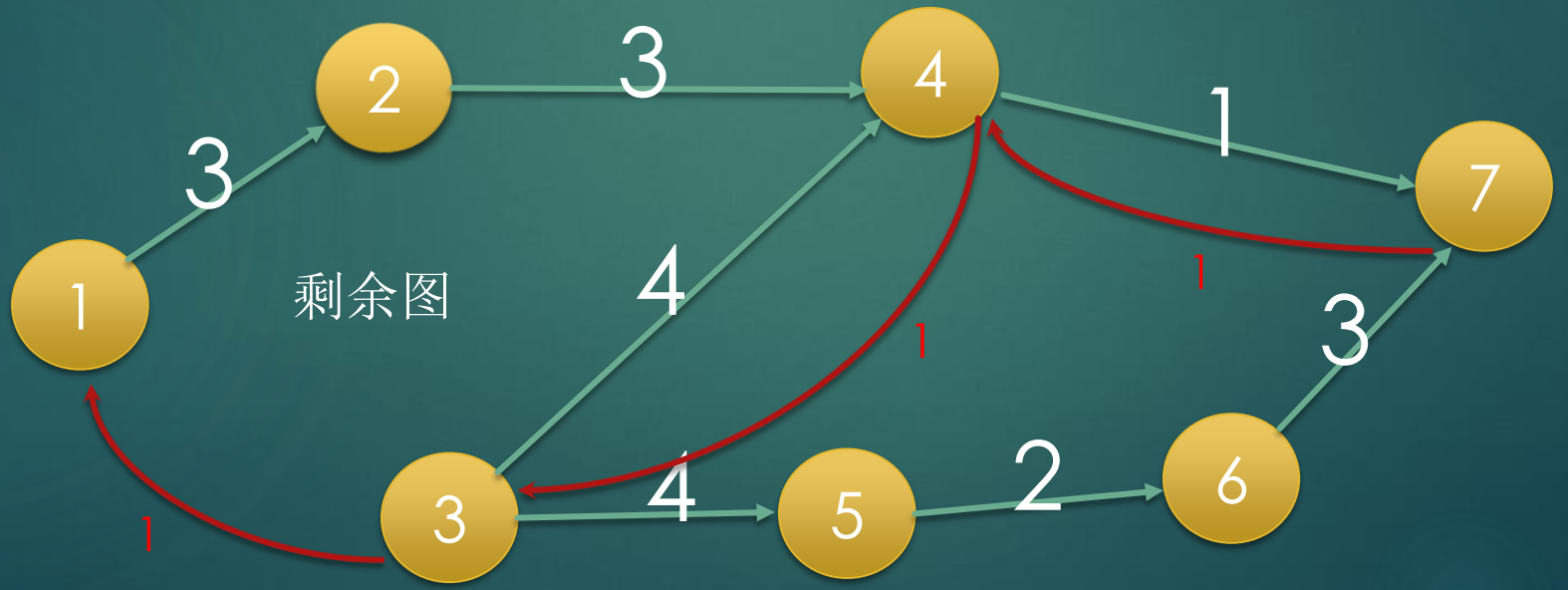


剩余图（残余网络）的概念

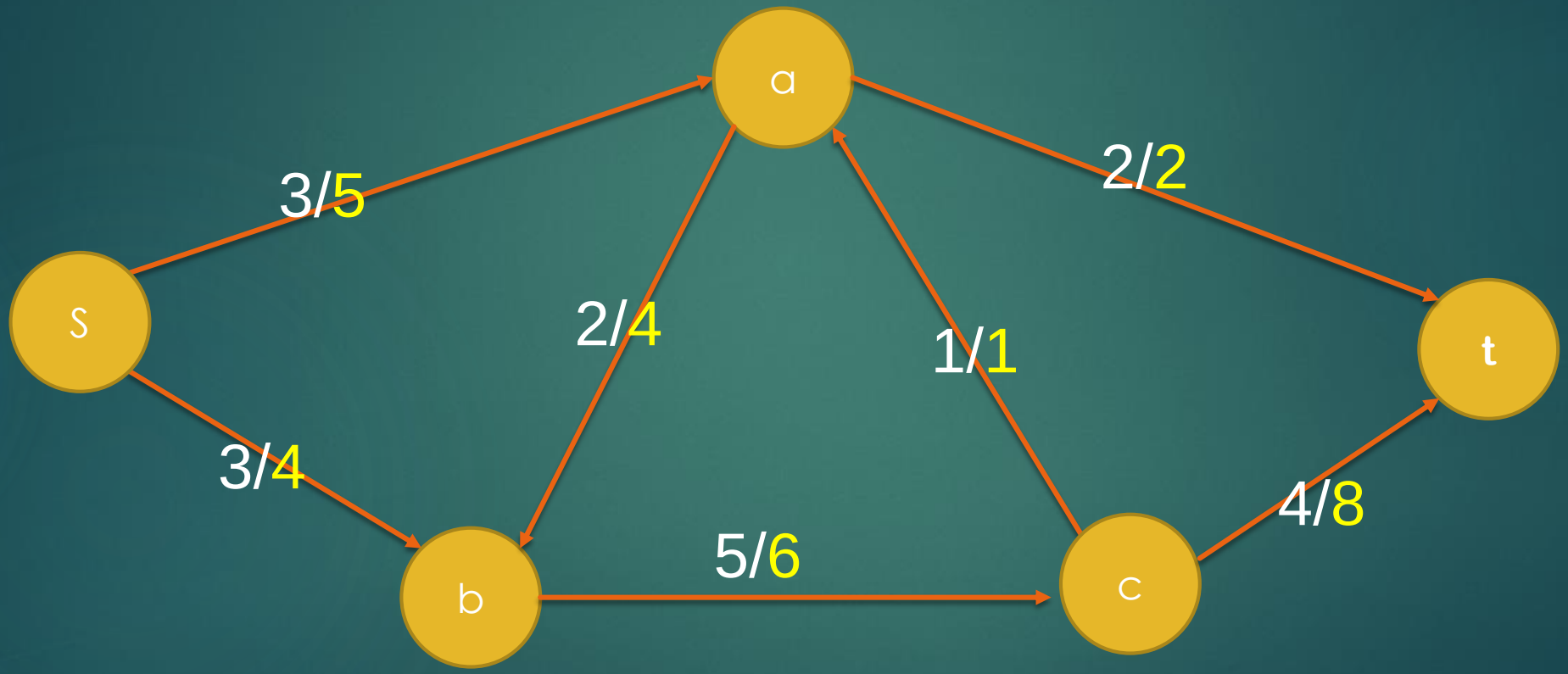
- 给定一个流量网络G(V,E)、源点s、汇点t、容量函数c，以及其上的流量函数f。我们这样定义对应的剩余图$G_1(V_1,E_1)$：剩余图中的点集与流量网络中的点集相同，即$V_1=V$。对于流量网络中的任一条边$(u,v) \in E$，若$f(u,v)<c(u,v)$，那么边$(u,v) \in E_1$，这条边在剩余图中的权值为$g(u,v)=c(u,v)-f(u,v)$；同时，若$f(u,v)>0$，那么边$(v,u) \in E_1$，这条边在剩余图中的权值为$g(v,u)=f(u,v)$。

例子

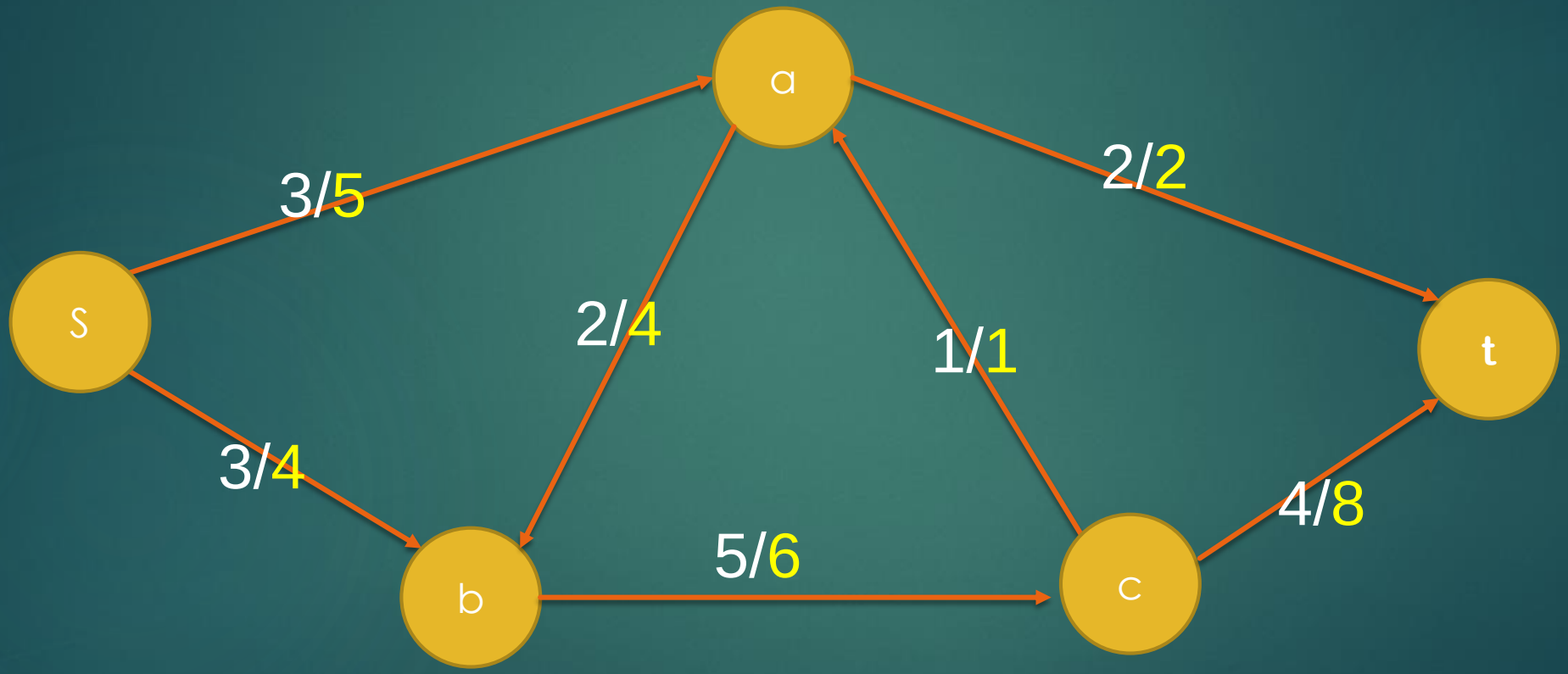




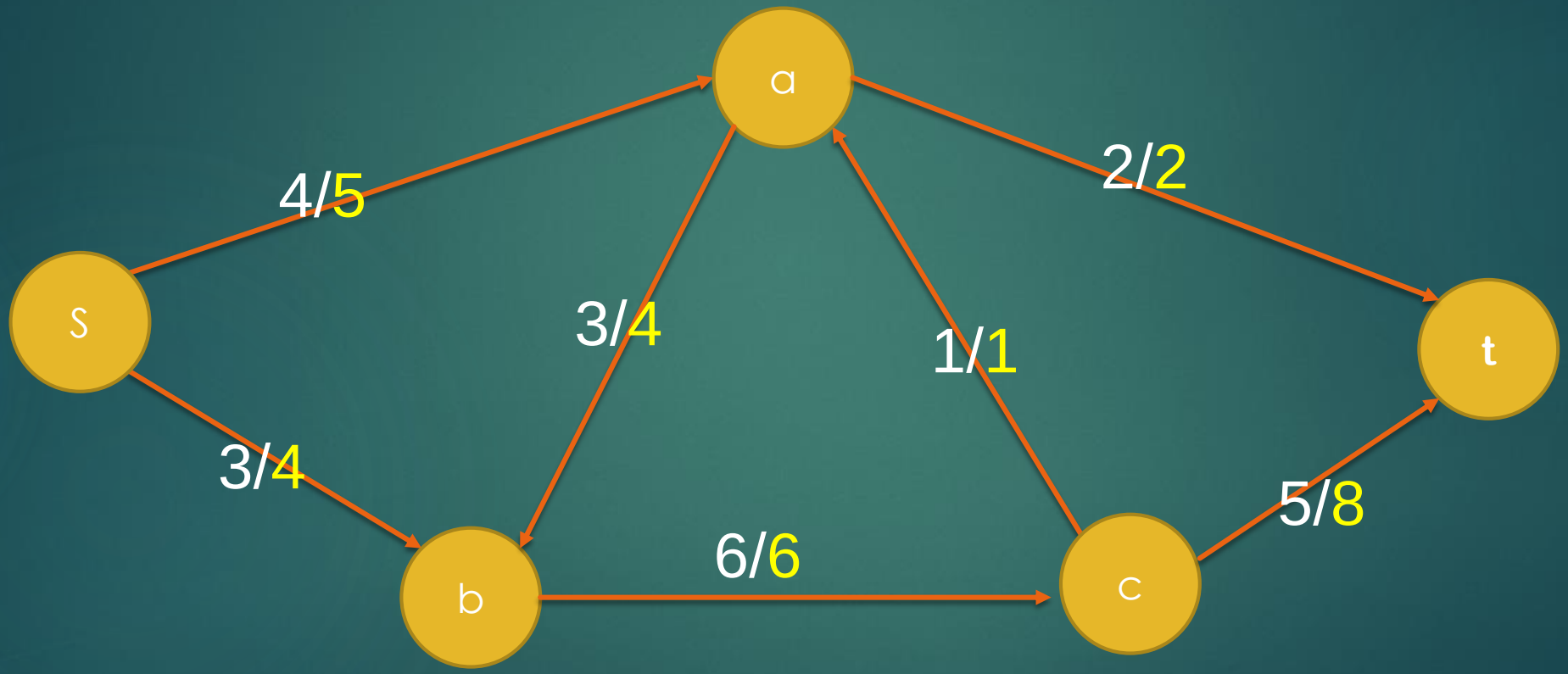
能否使整个图的总流量变大？



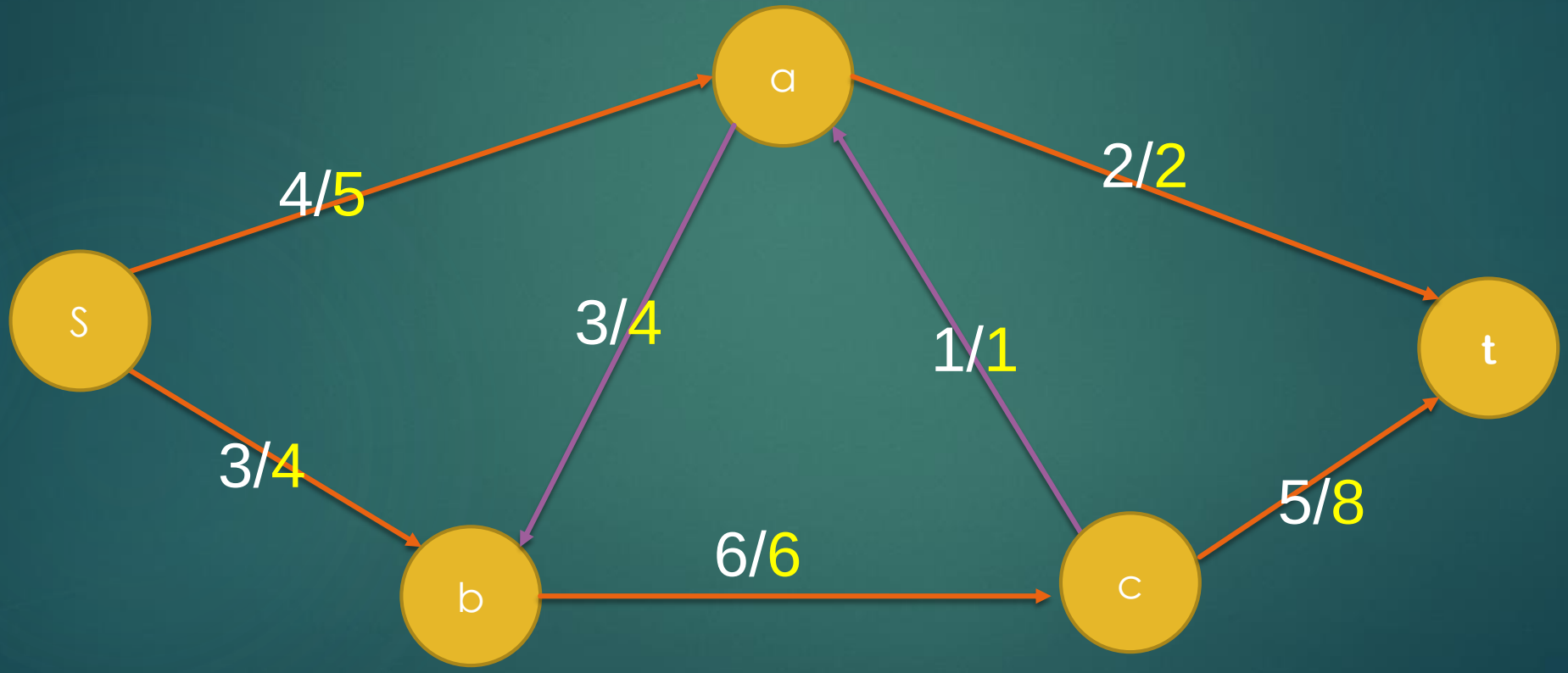
s → a → b → c → t把这条路径上的每条道路的实际流量都加 1



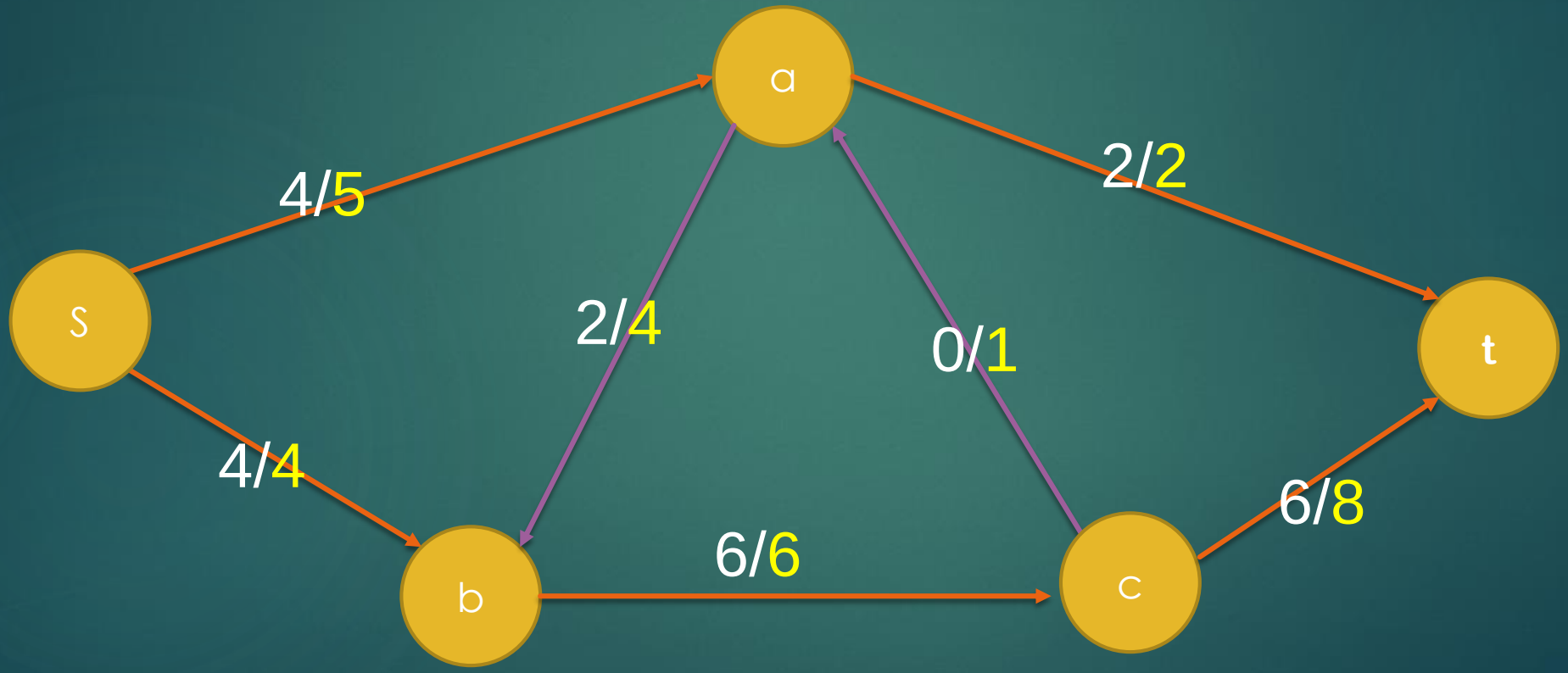
我们还能进一步增加流量吗？



- 考虑路径 s → b → a → c → t，注意这条路径中只有 s → b 段和 c → t 段是沿着道路方向走的，而 b → a 段和 a → c 段与图中所示的箭头方向正好相反。



- 我们把路径中所有与图中箭头方向相同的路段的实际流量都加 1 ，路径中所有与图中箭头方向相反的路段的实际流量都减 1 。



s → b → a → c → t

两种增加流量的模式

- 首先，从 s 点出发，寻找一条到 t 点的路径。
- 途中要么顺着某条流量还没满的（还能再加流量的）道路走一步，
- 要么逆着某条流量不为零的（能够减少流量的）道路走一步。
- 我们把这样的路径叫做“增广路径”。

推论

- 我们可以发现，流量网络中的每条边在剩余图中都化作一条或二条边。剩余图中的每条边都表示在原流量网络中能沿其方向增广。
- 剩余图的权值函数g(u,v)表示在流量网络中能够沿着u到v的方向增广大小为g(u,v)的流量。
- 所以在剩余图中，从源点到汇点的任意一条简单路径都对应着一条增广路，路径上每条边的权值的最小值即为能够一次增广的最大流量。

最大流定理

- 如果残留网络上找不到增广路径，则当前流为最大流；反之，如果当前流不为最大流，则一定有增广路径。

Ford-Fulkerson方法

1. 在剩余图中寻找源点到汇点的路径（增广路）
2. 如果不存在增广路，算法结束。否则，沿着该路径尽可能的增加流量，返回第一步。

Dfs实现Ford-Fulkerson方法


```
struct  edge{
    int to, cap, rev ;
};
vector<edge> G[MAXV] ;
bool used[MAXV] ;
void add_edge(int from, int to, int cap){
    G[from].push_back((edge){to, cap, G[to].size()}) ;
    G[to].push_back((edge){from, 0, G[from].size()-1}) ;
}
```

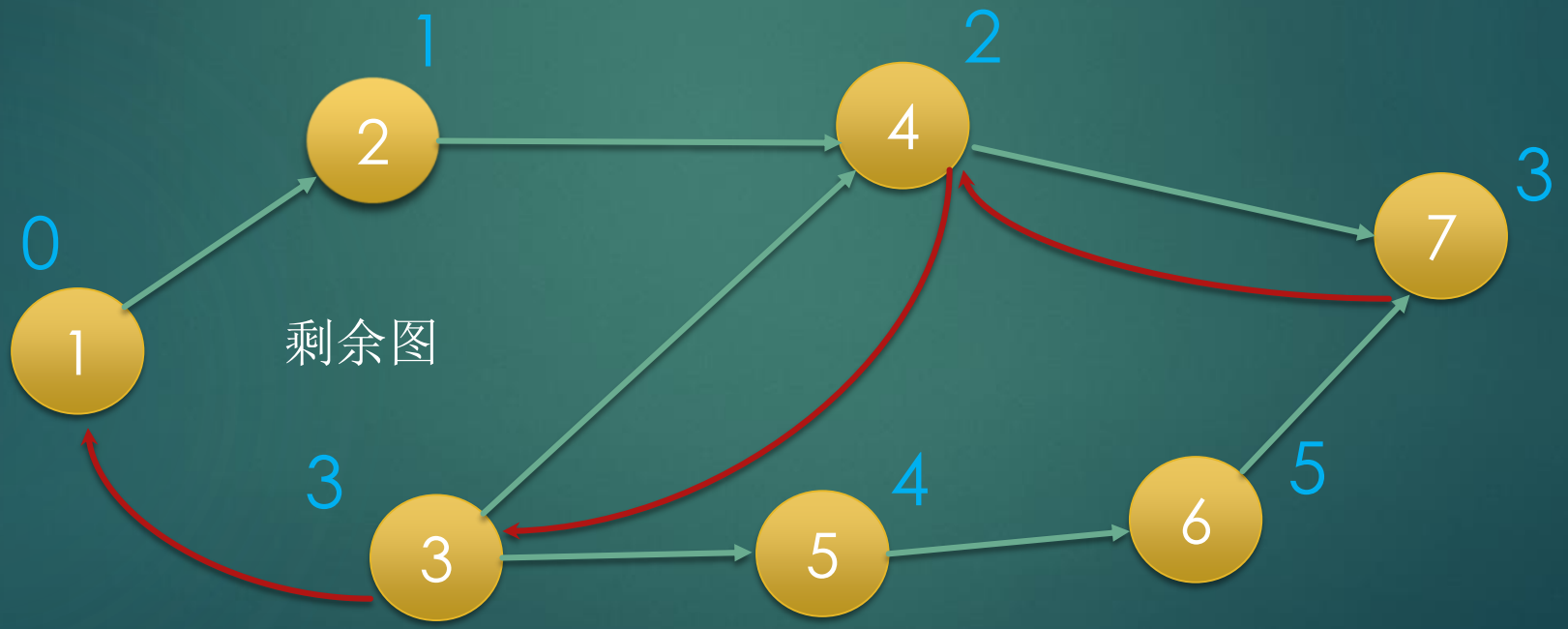
Dfs实现Ford-Fulkerson方法


```
int dfs(int v, int t, int f){
    if(v==t)return f;
    used[v] = true ;
    for(int i=0; i<G[v].size(); i++){
        edge&  e = G[v][i] ;
        if(!used[e.to] && e.cap >0){
            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 ;  }
            }
        }
        return 0;
    }
```

```
int max_flow(int s, int t){
    int flow = 0 ;
     while(true){
        memset(used , 0, sizeof(used)) ;
          int f = dfs(s, t, INF) ;
          if(f==0)return flow ;
          flow += f ;
          }
     }
```

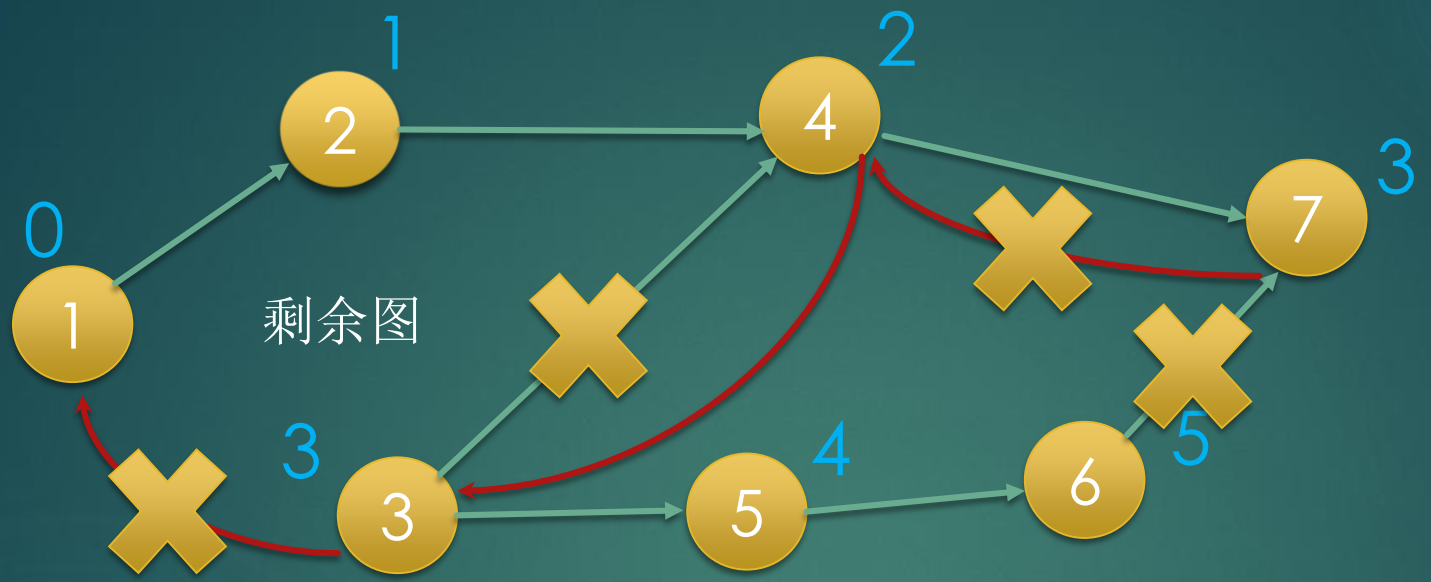
顶点的层次

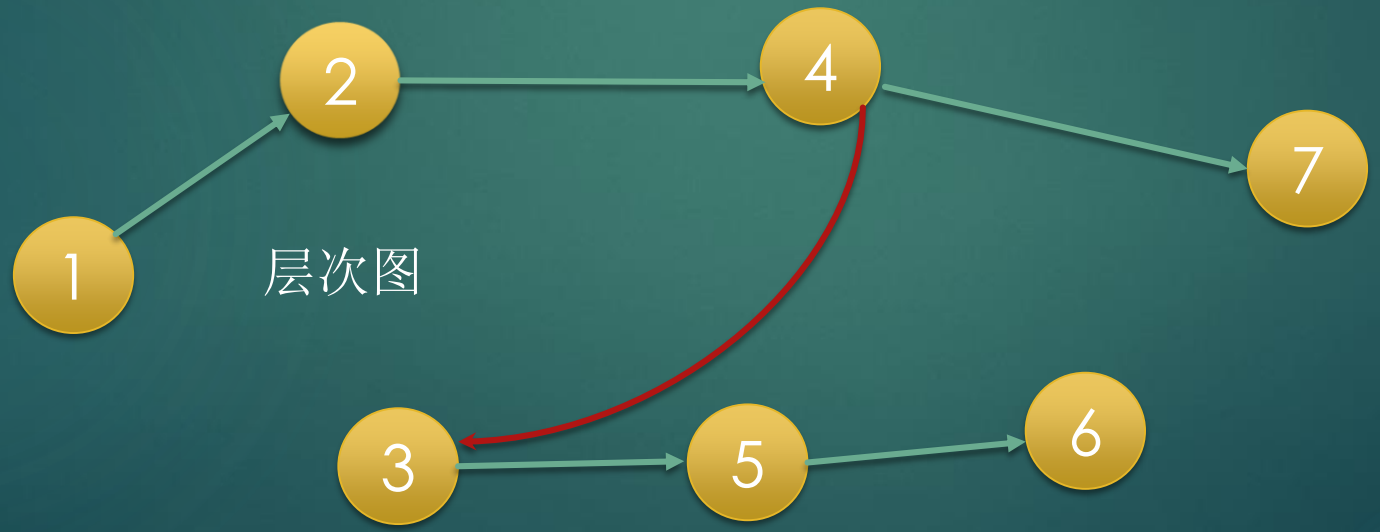
- 在剩余图中，我们把从源点到点u的最短路径长度称作点u的层次，记为level (u)，源点的层次为0。



层次图

- 我们这样定义层次图$G_2(V_2，E_2)$，对于剩余图$G_1(V_1,E_1)$中的一条边(u,v)，当且仅当level (u)+1= level (v)时，边$(u,v) \in E_2$；V_2等于E_2的盖点集。
- 直观地讲，**层次图是建立在剩余图基础之上的一张“最短路图”**。





当且仅当
level (u)+1=
level (v)时，
边(u,v) ∈ E_2

阻塞流

- 在流量网络中存在一可行流f，当该网络的层次图G_2中不存在增广路时，我们称流函数f为层次图G_2的阻塞流。

最短路径增值算法

1、初始化流量，计算出剩余图
2、根据剩余图计算层次图。若汇点不在层次图内，则算法结束
3、在层次图内不断用bfs增广，直到层次图内没有增广路为止
4、转步骤2

最短路径增值算法

- 算法中，2、3步被循环执行，我们将执行2、3步的一次循环称为一个**阶段**。每个阶段中，我们首先根据剩余图建立层次图，然后不断用bfs在层次图内增广，寻找阻塞流。增广完毕后，进入下一个阶段。这样不断重复，直到汇点不在层次图内出现为止。**汇点不在层次图内意味着在剩余图中不存在一条从源点到汇点的路径，即没有增广路。**

注意

- 在程序实现的时候，层次图并不用被“建”出来，我们只需对每个顶点标记层次，增广的时候，判断边是否满足level (u)+1= level (v)这一约束即可。

定理

- 对于有n个点的流量网络，在最短路径增值算法中，最多有n个阶段。

Dinic算法

- Dinic算法的思想也是分阶段地在层次图中增广。它与最短路径增值算法不同之处是：在Dinic算法中，我们用一个dfs过程代替多次bfs来寻找阻塞流。下面给出其算法步骤：

1、初始化流量，计算出剩余图。

2、根据剩余图计算层次图。若汇点不在层次图内，则算法结束。

3、在层次图内用一次dfs过程增广。

4、转步骤2。

Dfs增广过程


```
p←s ;
While outdegree(s)>0
    u ← p.top ;
    if u<>t
          if outdegree(u)>0
              设(u,v)为层次图中的一条边;
              p ← v ;
          else
              从p和层次图中删除点u,
              以及和u连接的所有边;
    else
        增广p（删除了p中的饱和边）;
        令p.top为p中从s可到达的最后顶点 ;
end while
```

邻接表结构


```
1. struct Edge
2. {
3.     int u, v, cap;
4.     Edge() {}
5.     Edge(int u, int v, int cap): u(u), v(v), cap(cap) {}
6. } es[MAXE];
7. vector<int>    tab[MAXN]; // 顶点表
```

构造剩余图


```
1. void addedge(int u, int v, int cap)
2. {
3.     tab[u].push_back(R);
4.     es[R++] = Edge(u, v, cap); // 正向边权值等于容量。
5.     tab[v].push_back(R);
6.     es[R++] = Edge(v, u, 0); // 反向边权值为0。
7.     // 正向边下标通过与1异或就得到反向边下标, 2 ^ 1 == 3 ; 3 ^ 1 == 2
8. }
```

构造剩余图


```
1. for (int i = 0; i < N; i++)
2.             tab[i].clear();
3. for (int i = 0; i < M; i++){
4.             int u, v, cap;
5.             scanf(" (%d,%d)%d", &u, &v, &cap);
6.             addedge(u, v, cap);
7.         }
```

构建层次图


```
1.  int BFS()
2.  {
3.      queue<int> q ;
4.      q.push(S) ; memset(dis, 0x3f, sizeof(dis)) ; dis[S] = 0 ;
5.      while (!q.empty()) {
6.          int h = q.front() ;  q.pop() ;
7.          for (int i = 0; i < tab[h].size(); i++){
8.              Edge &e = es[tab[h][i]];
9.              if (e.cap > 0 && dis[e.v] == 0x3f3f3f3f){
10.                 dis[e.v] = dis[h] + 1;
11.                 q.push(e.v);  }
12.         }
13.     }
14.     return dis[T] < 0x3f3f3f3f; // 返回是否能够到达汇点
15. }
```

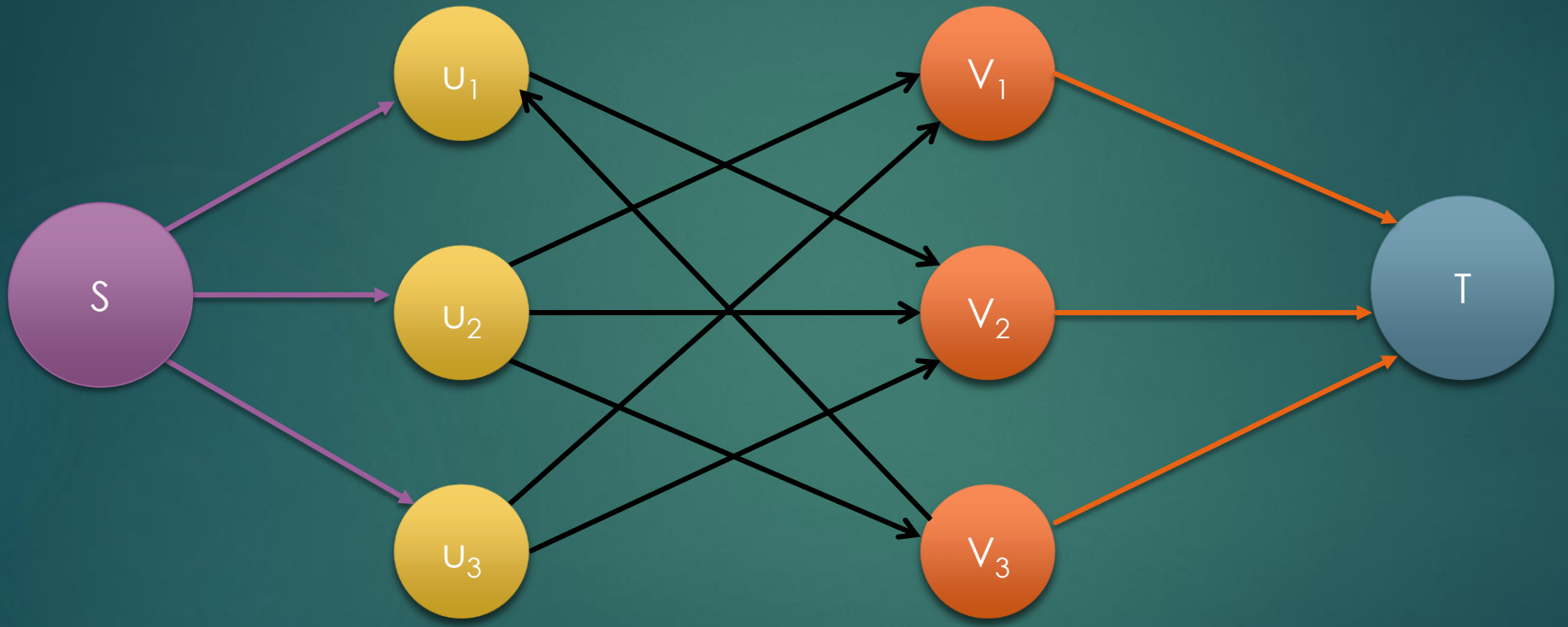
```
1.  int Dinic(int x, int maxflow){
2.      if (x == T) return maxflow ;
3.     for (int i = current[x]; i < tab[x].size(); i++) { // i = current[x] 当前弧优化。
4.          current[x] = i ;
5.          Edge &e = es[tab[x][i]] ;
6.          if (dis[e.v] == dis[x] + 1 && e.cap > 0) {
7.              int flow = Dinic(e.v, min(maxflow, e.cap)) ;
8.              if (flow) {
9.                  e.cap -= flow ; // 正向边流量降低。
10.                 es[tab[x][i] ^ 1].cap += flow ; // 反向边流量增加。
11.                 return flow ;  }
12.         }
13.     }
14.     return 0; // 找不到增广路，  退出。
15. }
```

Dinic算法

Dinic算法流程


```
1. int DINIC_MAIN( ){
2.     int ans = 0;
3.     while (BFS()) { // 建立分层图
4.         int flow ;
5.         memset(current, 0, sizeof(current)); // BFS后应当清空当前弧数组。
6.          while (flow = Dinic(S, 0x3f3f3f3f)) // 一次BFS可以进行多次增广。
7.              ans += flow; }
8.       return ans;
9.  }
```

二分图的最大匹配问题

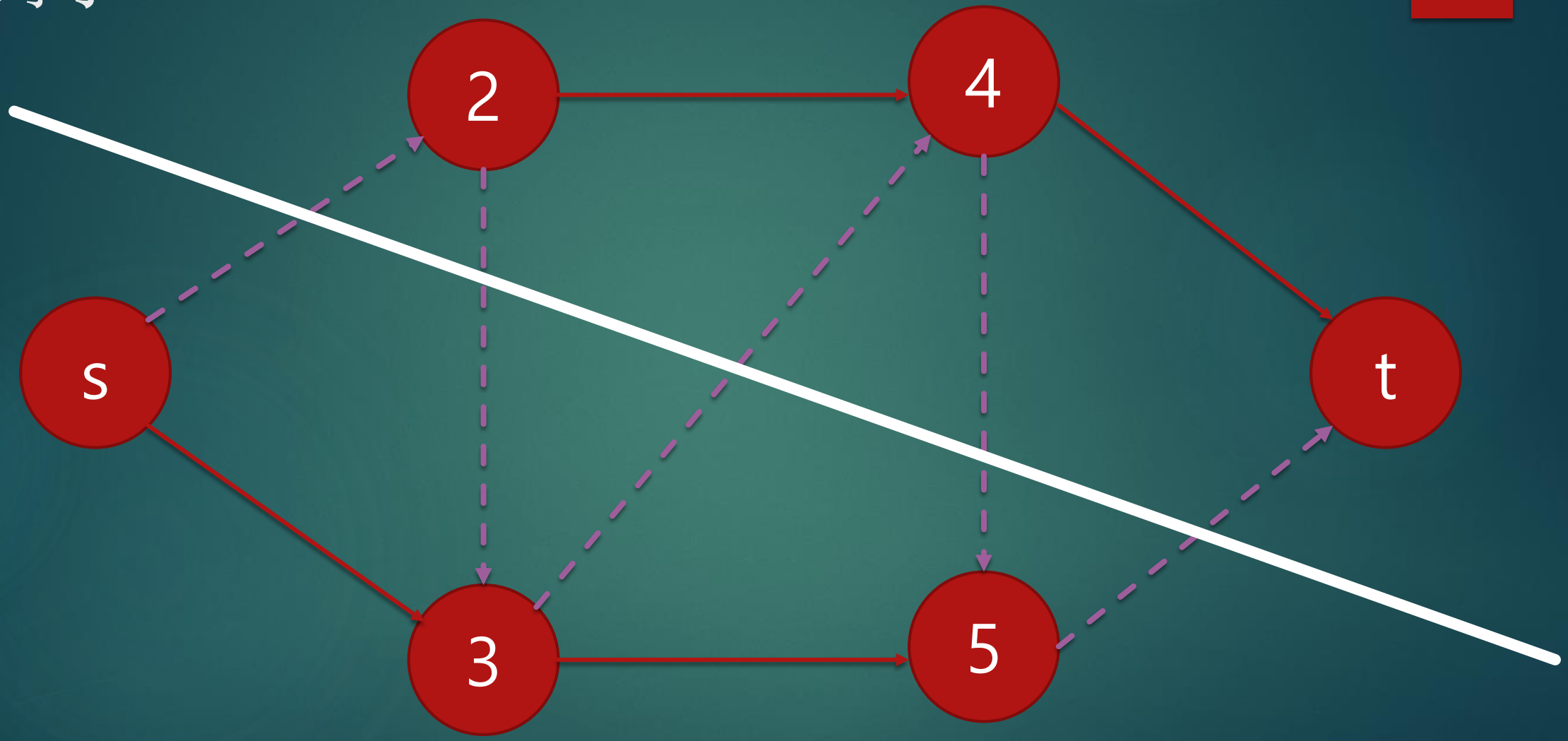


最大流与最小割

- 流网络G(V,E)的**割**（cut）将点集V 划分为 S和T （ T = V− S）两个部分，使得源 $s \in S$且汇$t \in T$ 。
- 符号[S, T]代表一个边集合 $\{< u, v > | < u, v > \in E, u \in S, \mathrm{v} \in T\}$，穿过割[S, T]的净流（net flow）定义为 f(S,T) ， 割[S, T]的容量（capacity）定义为c(S,T) ，
- 一个网络的最小割（minimum cut）也就是该网络中容量最小的割。

例子

割[T，S] ={ <2,3>,<4,5>}



割[S，T] ={ <s,2>,<3,4>,<5,t>}

引理 割与流的关系

- 在一个流网络G(V,E)中，设其任意一个流为f，且[S，T]为 G一个割。则通过割的净流为f(S,T) =|f| 。

推论 对偶问题的性质

- 在一个流网络G(V,E)中，设其任意一个流为f，任意一个割为[S，T]。必有|f|≤ c[S,T]。

定理 最大流最小割定理

- 如果f 是具有源s和汇t的流网络G(V,E)中的一个流，则下列条件是等价的：

1. f 是G 的一个最大流。
2. 残留网络G_f不包含增广路径。
3. 对G的某个割[S，T]，有|f|= c[S,T]。