

HIT
CS&E

第七章 MaxMin方法

骆吉洲
计算机科学与技术学院

HIT
CS&E

提纲

- 7.1 网络流算法
 - 7.1.1 流网络和流
 - 7.1.2 Ford-Fulkson算法
 - 7.1.3 推送复标算法
 - 7.1.4 复标前置算法
- 7.2 匹配算法
 - 7.2.1 匹配与覆盖
 - 7.2.2 最大二分匹配算法
 - 7.2.3 最大加权二分匹配算法
 - 7.2.4 稳定匹配算法
- 7.3 ... 补充阅读材料

HIT
CS&E

教学目的

难点: Max-Min关系及其在算法设计和分析中的应用

重点: 基本算法层面: (1)最大流-最小割算法
(2)最大匹配-最小覆盖算法
(3)基本图论算法的总结和复习

算法设计技术层面: (1)Max-min方法
(2)精益求精的算法设计过程

问题特征分析能力层面: (1)准确, 渐进

参考书和最新文献:

1. 网络流: 理论、算法和应用. 机械工业出版社, 2004
2. Incremental graph pattern matching, VLDB, 2011
3. On the complexity of view update and application in annotation propagation. TKDE, 2012

HIT
CS&E

7.1 Maximum Flow

- 7.1.1 流网络与流
- 7.1.2 Ford-Fulkerson方法
- 7.1.3 推送复标算法
- 7.1.4 复标前置算法

HIT
CS&E

7.1.1 流网络与流

构建和使用各种网络时, 需要考虑网络的**通行能力**
网络的**通行能力**受网络结构的限制

- 道路的宽窄
- 管道的粗细
-

流网络的种类

- 英特网
- 电话网
- 高速路网
- 铁路网
- 电网
- 输气网络
- 排水网络
- 输水网络

能否建立**通用模型**来研究网络的**最大通行能力**呢?

HIT
CS&E

流网络

加权有向图 $G=(V,E)$

- 顶点表示网络中的关键节点
- 边 uv 表示 u 和 v 之间的连接, 容量 $c(u,v) \geq 0$ 表示边的通行能力;
如果 $uv \notin E$, 则定义容量 $c(u,v)=0$
- 两个特殊顶点 s 和 t , s 称为**源(source)**, t 称为**汇(sink)**
- G 中每个顶点均位于某条由 s 到 t 的路径上($|E| \geq |V|-1$)

流网络 $G=(V,E)$ 上的一个流

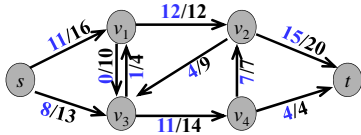
流

– 是一个实值函数 $f: V \times V \rightarrow \mathbb{R}$, 满足

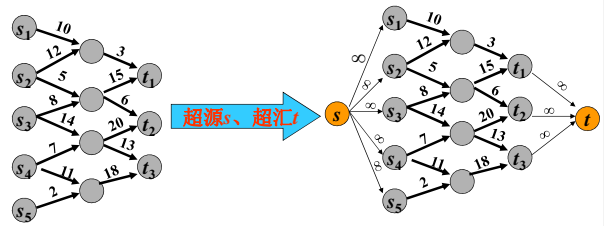
- 容量约束: $f(u,v) \leq c(u,v)$ 对 $\forall uv \in E$ 成立
- 反对称性: $f(u,v) = -f(v,u)$
- 守恒约束: $\sum_{v \in V} f(u,v) = 0$ 对任意 $u \in V - \{s,t\}$ 成立

– $f(u,v)$ 称为顶点 u 到顶点 v 的流量, 可为正、负、零值

– 流 f 的值定义为 $|f| = \sum_{v \in V} f(s,v)$



多源多汇的网络



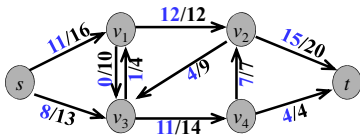
只需讨论单源单汇的网络流



最大流问题

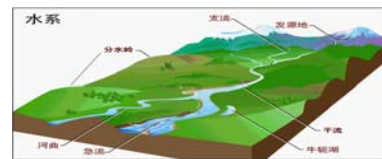
给定流网络 $G=(V,E)$, 其源为 s , 汇为 t , 找出从 s 到 t 的最大流。

• 怎样高效率地找?



直观认识

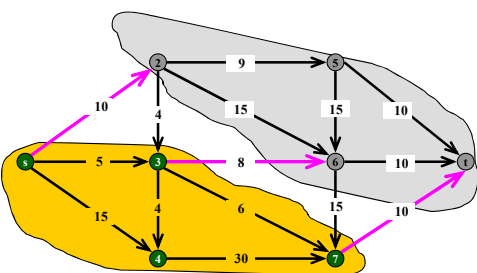
河水的最大流量取决于
干流中河道狭窄处的通行能力



这种观察能否用于研究最大流问题呢?



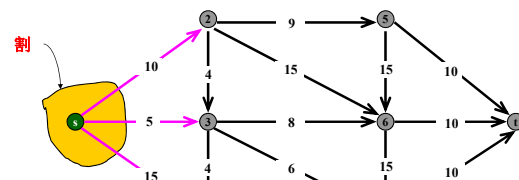
从 s 流到 t 的最大流量不会超过 $10+8+10=28$



割

给定流网络 $G=(V,E)$, 其源为 s , 汇为 t

G 的一个割 (cut) 是 V 的 2-集合划分 S, T ($T=V-S$) 使得 $s \in S, t \in T$



割的容量定义为

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

Capacity = 30

最小割问题

给定流网络 $G=(V,E)$, 其源为 s , 汇为 t , 找出流网络 G 中容量最小的割。

- 怎样高效率地找?

Capacity = 28

弱对偶关系 (预备)

顶点集之间的流量

- 设 $X, Y \subseteq V$ 记 $f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$
- 称 $f(X, Y)$ 为顶点集 X 和 Y 之间的流量
- 守恒约束等价于 $f(u, V) = 0$ 对 $\forall u \in V - s - t$
- 上述表示经常用来简化网络流涉及的等式

引理1

设 f 是流网络 $G=(V,E)$ 中的一个流, 则

- $f(X, X) = 0$ 对 $\forall X \subseteq V$ 成立
- $f(X, Y) = -f(Y, X)$ $\forall X, Y \subseteq V$ 成立
- $\forall X, Y, Z \subseteq V$ 且 $X \cap Y = \emptyset$, 有

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$

$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$$

故, $|f| = f(s, V) = f(s, V) - f(V - s, V) = -f(V - s, V) = f(V, V - s) = f(V, t) + f(V, V - s - t) = f(V, t)$

最大流—最小割间弱对偶关系

引理2

给定流网络 $G=(V,E)$, s 是源, t 是汇. 设 f 是 G 上的一个流, S, T 是 G 的一个割, 则 $f(S, T) = |f|$.

证明: $f(S, T) = f(S, V) - f(S, S)$
 $= f(S, V)$
 $= f(s, V) + f(S - s, V)$
 $= f(s, V)$
 $= |f|$

引理1第3部分
 引理1第1部分
 引理1第3部分
 $f(S - s, V) = 0$

引理3

给定流网络 $G=(V,E)$. 设 f 是 G 上的一个流, S, T 是 G 的一个割, 则 $|f| \leq c(S, T)$

$|f| = f(S, T)$

$$= \sum_{u \in S} \sum_{v \in T} f(u, v)$$

$$\leq \sum_{u \in S} \sum_{v \in T} c(u, v)$$

$$= c(S, T)$$

最大流—最小割间的弱对偶关系

引理

给定流网络 $G=(V,E)$. 设 f 是 G 上的任意一个流, S, T 是 G 的任意一个割, 则 $|f| \leq c(S, T)$

怎么用这个max-min关系来高效地求出最大流 (最小割) 呢?

利用max-min关系设计最大流算法

- 初始化一个可行流 f
 - ≥ 0 -流: 所有边的流量均等于0的流
- 不断将 f 增大, 直到 f 不能继续增大为止
- 找出一个割 S, T 使得 $|f| = c(S, T)$
 - 由此断言 f 是最大流, 而 S, T 是最小割

max-min关系提供了高效求解最大流-最小割问题的机制!
 关键在于第2步怎么做?

7.1.2 Ford-fulkerson方法

算法的基本思想

- 对于当前的流, ...
- ...找出一条从 s 到 t 的路径 p (增广路径) 和正数 $a > 0$, 使得 p 上的每条边 uv 的流量增加 a 之后仍然满足容量约束, 即 $f(u, v) + a \leq c(u, v)$
- 则将 p 上每条边的流量增加 a , 得到一个更大的流

怎么增大当前的流?

Ford-fulkerson算法概要

算法Ford-Fulkerson(G, s, t)

Input 流网络 G , 源 s , 汇 t

Output G 中从 s 到 t 的最大流

- 1 初始化所有边的流量为0
- 2 while 存在增广路径 p do
- 3 沿路径 p 增大流量得到更大的流 f
- 4 return f

- 增广路径如何找?
- 增广路径上可以增加的流量有多大?
- 该方法总能找到最大流吗?

剩余网络(Residual Network)

增广路径如何找?

我们擅长找路径

寻找增广路径能否转变为在某个图中找路径呢?

$E_f = \{uv: c_f(u,v) = c(u,v) - f(u,v) > 0\}$

剩余网络(形式定义)

增广路径如何找?

- 增广路径是Residual network中从 s 到 t 的路径
 - Residual capacities: $c_f(u,v) = c(u,v) - f(u,v)$
 - Residual network: $G_f = (V, E_f)$, 其中

$$E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$$
 - $f(u,v) < c(u,v)$, 则 $c_f(u,v) = c(u,v) - f(u,v) > 0$, $(u,v) \in E_f$
 - $f(u,v) > 0$ 则 $c_f(v,u) = c(v,u) - f(v,u) > 0$, $(v,u) \in E_f$
 - $c(u,v) = c(v,u) = 0$, 则 $f(u,v) = f(v,u) = 0$, 进而 $c_f(u,v) = c_f(v,u) = 0$
- 注意: E_f 中的边要么是 E 中的边, 要么是 E 中边的反向边: $|E_f| \leq 2|E|$
- Residual Network本身也可以看成是流网络

利用剩余网络找增广路径

利用剩余网络找增广路径

增广路径的剩余容量

增广路径上可以增大多少流量?

- P 是 G_f 中的一条增广路径
 - 其剩余容量 $c_f(p) = \min \{c_f(u,v) : (u,v) \text{ 是路径 } p \text{ 上的边}\}$
- 增广过程: 对路径 p 上的每条边 uv
 - 要么在边 uv 的流量上增加 $c_f(p)$, 即 $f(u,v) = f(u,v) + c_f(p)$
 - 要么在边 vu 的流量上减去 $c_f(p)$, 即 $f(v,u) = f(v,u) - c_f(p)$
 - 具体属于哪种情况, 根据 G
- 增广过程完成后, 得到值更大的流

HIT CS&E

Ford-Fulkerson算法

算法Ford-Fulkerson(G, s, t)

Input 流网络 G , 源 s , 汇 t

Output G 中从 s 到 t 的最大流

1. For $\forall uv \in E[G]$ do
2. $f(u, v) \leftarrow 0$
3. $f(v, u) \leftarrow 0$
4. While G_f 存在增广路径 p do
5. $c_f(p) = \min\{c_f(u, v) \mid uv \text{ 是 } p \text{ 上的边}\}$
6. For p 上的每条边 uv do
7. If uv 是流网络中的边 Then
8. $f(u, v) \leftarrow f(u, v) + c_f(p)$
9. $f(v, u) \leftarrow -f(u, v)$
10. Else
11. $f(v, u) \leftarrow f(v, u) - c_f(p)$
12. $f(u, v) \leftarrow -f(v, u)$
13. 修改剩余中网络 G_f 相应的边

HIT CS&E

Ford-fulkerson算法的正确性

引理4 给定流网络 $G=(V, E)$, s 是源, t 是汇; f 是 G 上的一个流. G_f 是流 f 在 G 上导出的Residual Network, f' 是 G_f 上的一个流. 则 $f+f'$ 是 G 上值为 $|f|+|f'|$ 的流.

其中, $(f+f')(u, v) = f(u, v) + f'(u, v)$

证明: (作业)

- (1)验证反对称性
- (2)验证容量约束
- (3)验证守恒约束
- (4)验证 $|f+f'| = |f| + |f'|$

HIT CS&E

Ford-fulkerson算法的正确性

引理4 (最大流-最小割定理) 给定流网络 $G=(V, E)$, s 是源, t 是汇; f 是 G 上的一个流. 则下列论断等价.

- (1) f 是最大流;
- (2) G_f 中不存在增广路径
- (3)对 G 的某个割 (S, T) , $|f| = c(S, T)$

证明: (1) \Rightarrow (2) 反证. 设 p 是 G_f 是最大流 f 对应的增广路径, 其剩余容量为 f_p . 由引理4知道, $f+f_p$ 是一个值比 $|f|$ 的流. 这与 f 是最大流矛盾.

(2) \Rightarrow (3) 由于 G_f 中没有从 s 到 t 的路径, 定义 $S = \{v \mid G_f \text{ 中存在从 } s \text{ 到 } v \text{ 的路径}\}$, $T = V - S$. 显然 $s \in S$ 且 $t \in T$. $\forall u \in S$ 且 $v \in T$, $f(u, v) = c(u, v)$, 否则 $uv \in E_f$ 进而导致 $v \in S$. 于是 S, T 是 G 的一个割. 由引理3知道 $|f| = f(S, T) = c(S, T)$

(3) \Rightarrow (1) 引理2表明 $|f| \leq c(S, T)$, 故 $|f| = c(S, T)$ 表明 f 是最大流.

HIT CS&E

复杂性(1)

算法Ford-Fulkerson(G, s, t)

Input 流网络 G , 源 s , 汇 t

Output G 中从 s 到 t 的最大流

1. For $\forall uv \in E[G]$ do
2. $f(u, v) \leftarrow 0$
3. $f(v, u) \leftarrow 0$
4. While G_f 存在增广路径 p do
5. $c_f(p) = \min\{c_f(u, v) \mid uv \text{ 是 } p \text{ 上的边}\}$
6. For p 上的每条边 uv do
7. If uv 是流网络中的边 Then
8. $f(u, v) \leftarrow f(u, v) + c_f(p)$
9. $f(v, u) \leftarrow -f(u, v)$
10. Else
11. $f(v, u) \leftarrow f(v, u) - c_f(p)$
12. $f(u, v) \leftarrow -f(v, u)$
13. 修改剩余中网络 G_f 相应的边

$O(|E_f|) = O(|E|)$

$O(|p|) = O(|E|)$

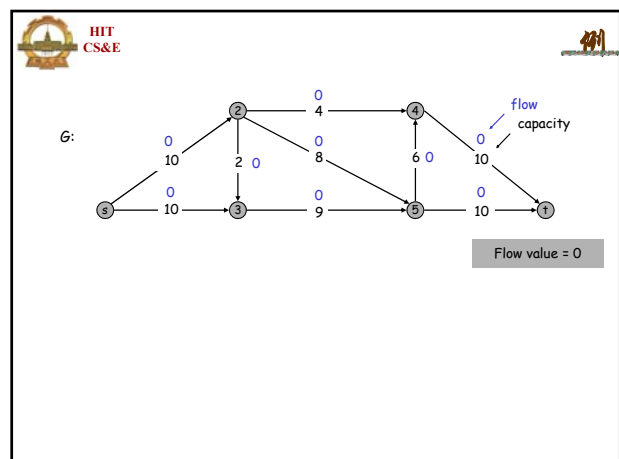
HIT CS&E

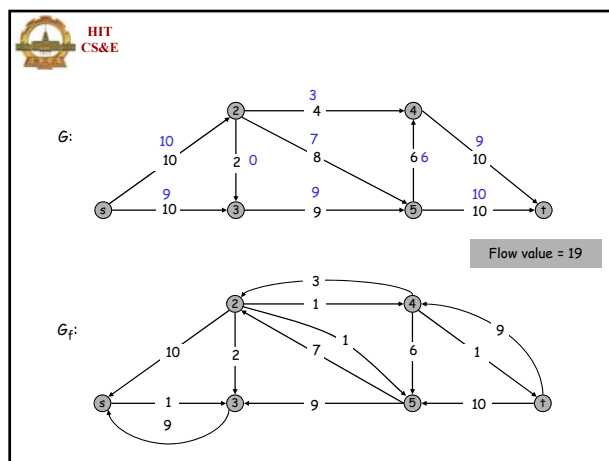
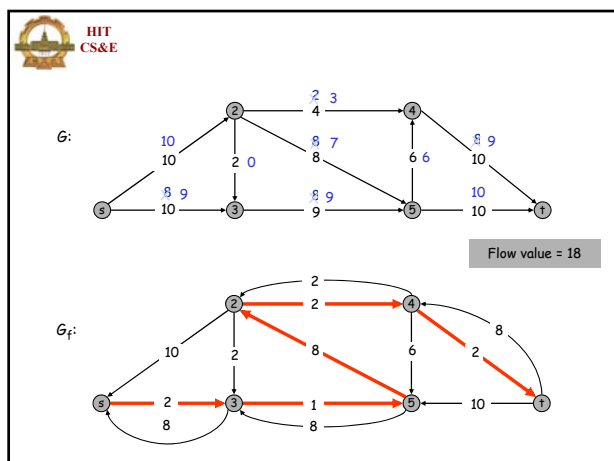
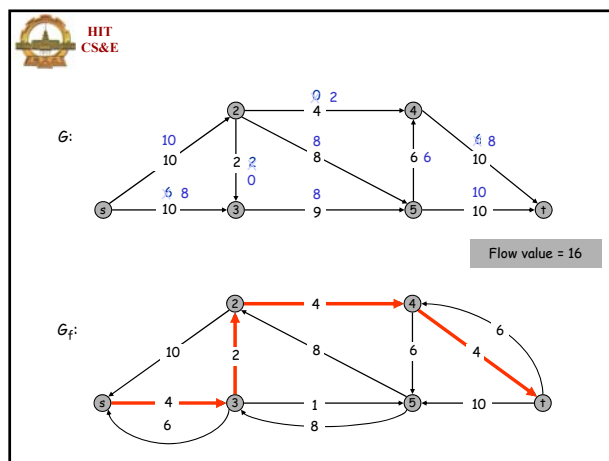
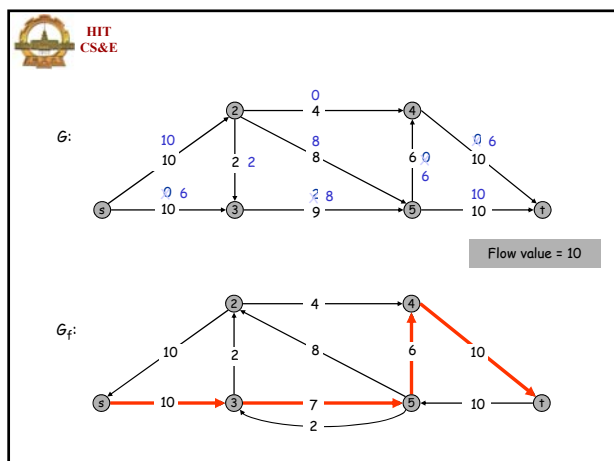
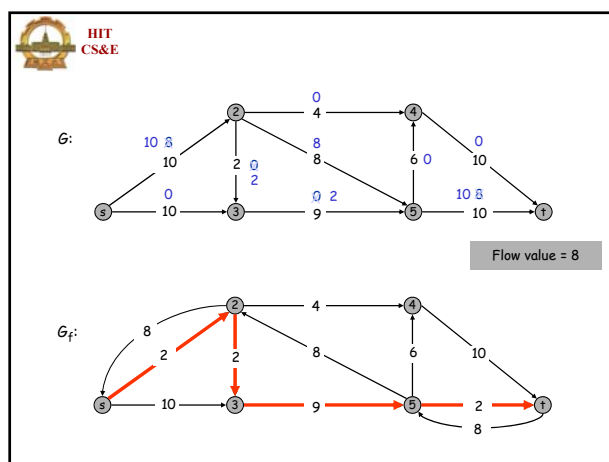
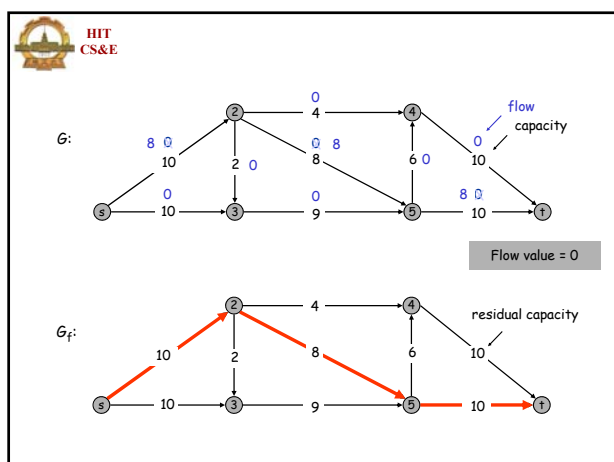
复杂性(2)

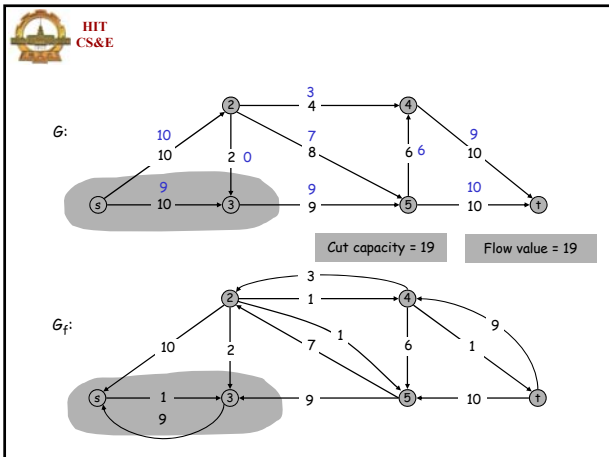
Ford-Fulkerson算法的时间复杂度 $O(|f^*||E|)$

怎么改进Ford-Fulkerson算法呢?

1. 选用好的路径计算方法
 - 明确选用BFS (Edmonds-Karp算法)
2. 计算特殊的增广路径
 - 第5步: 剩余容量 $c_f(p)$ 达到最大值(习题7.9)
3. 利用“最大流等于最小割”这一结论重新设计其他算法
 - 7.1.3节和7.1.4节



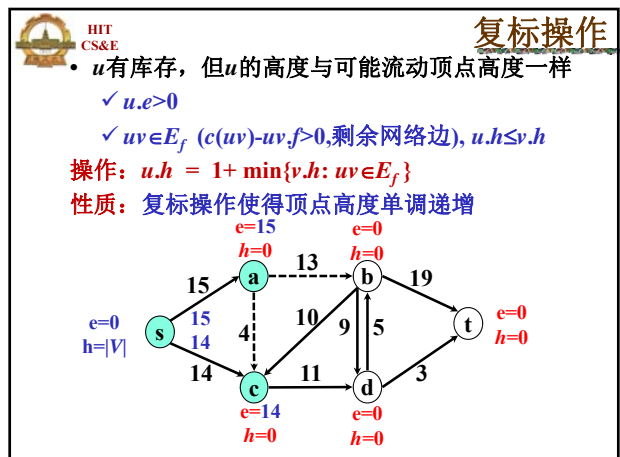
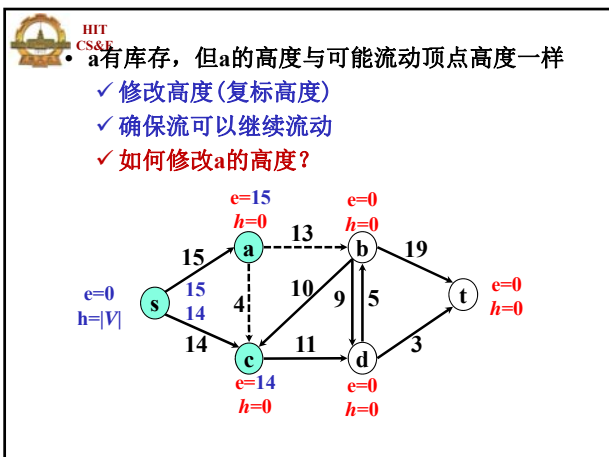
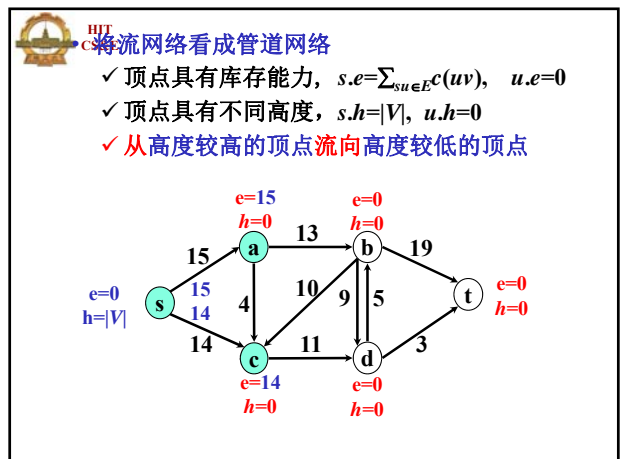
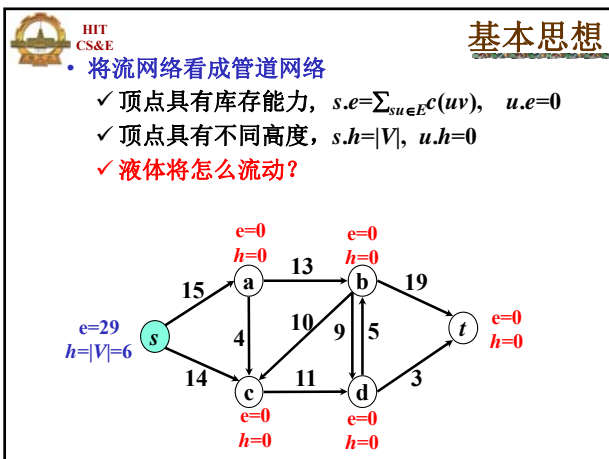


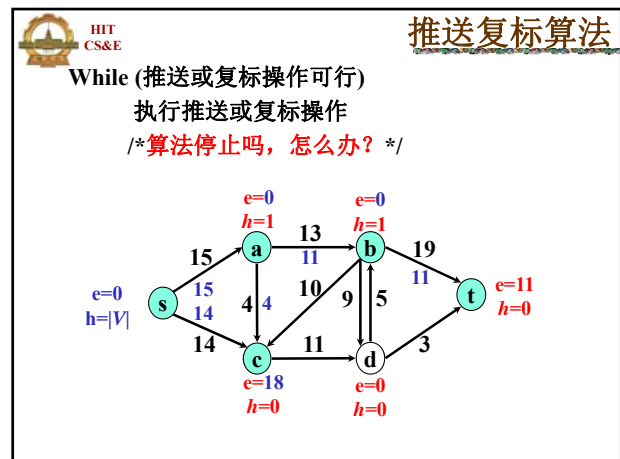
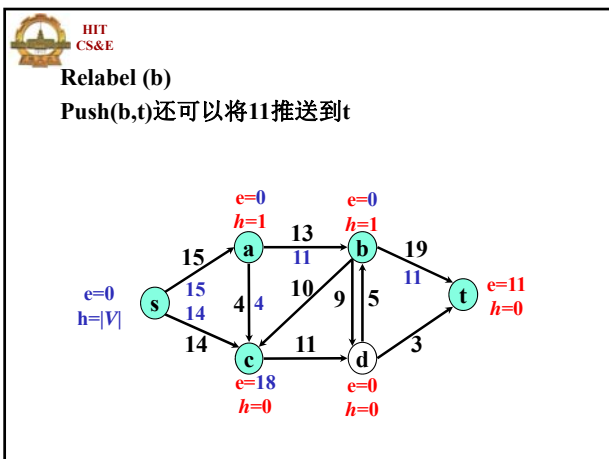
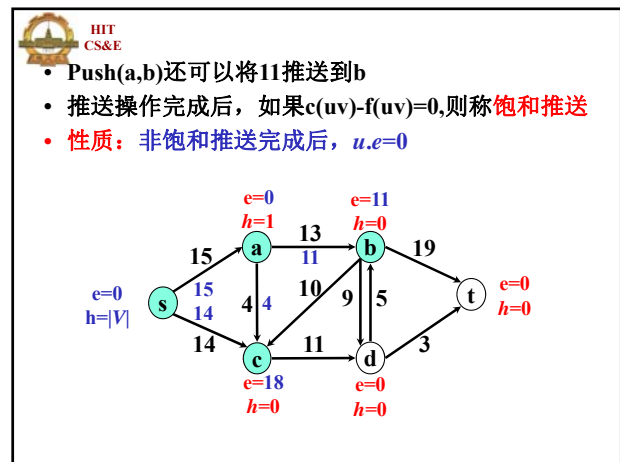
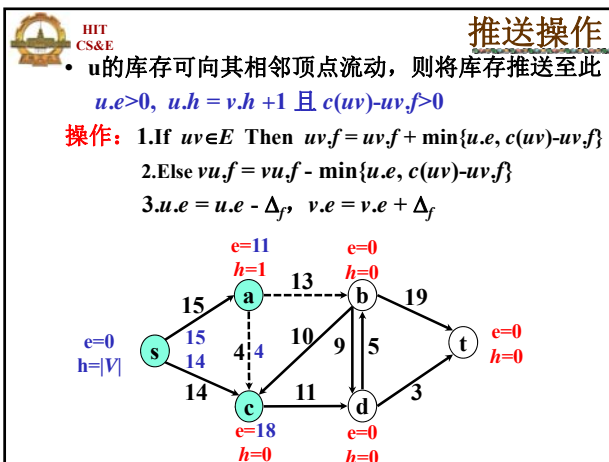
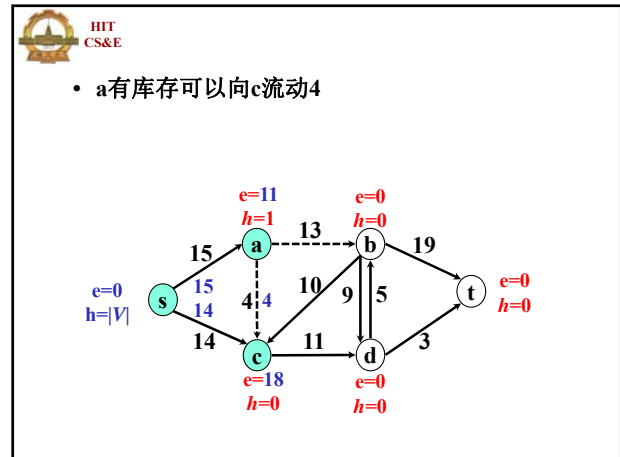
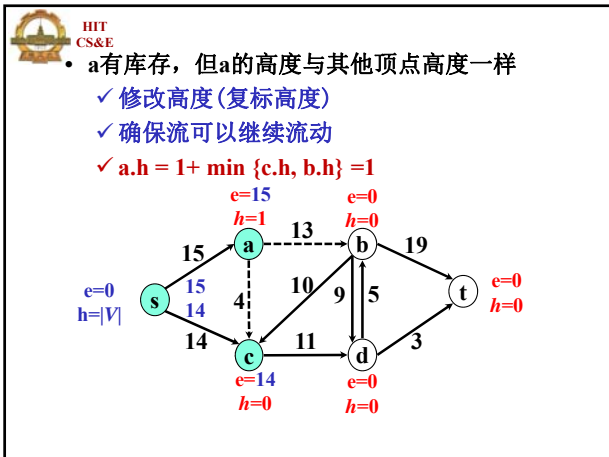


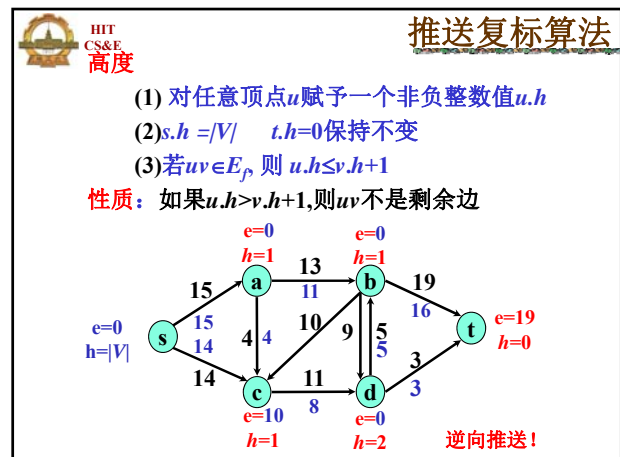
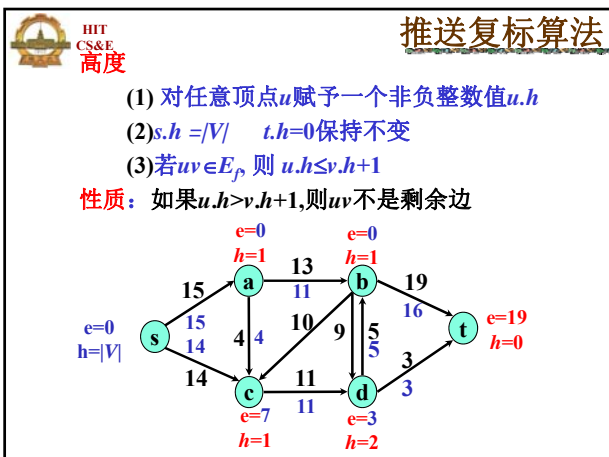
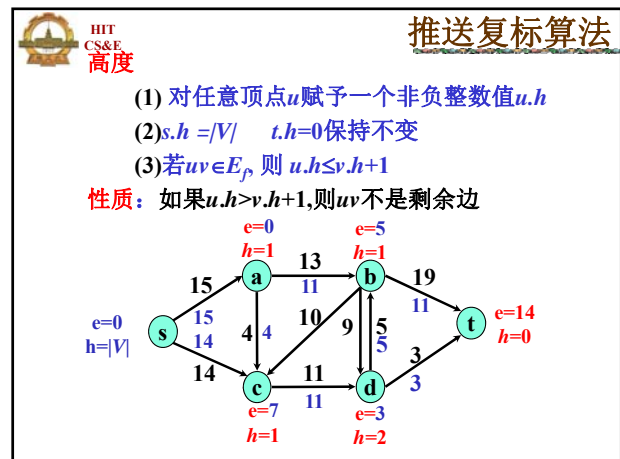
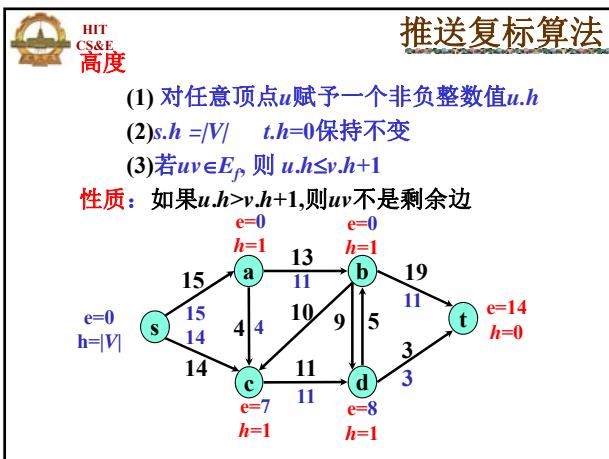
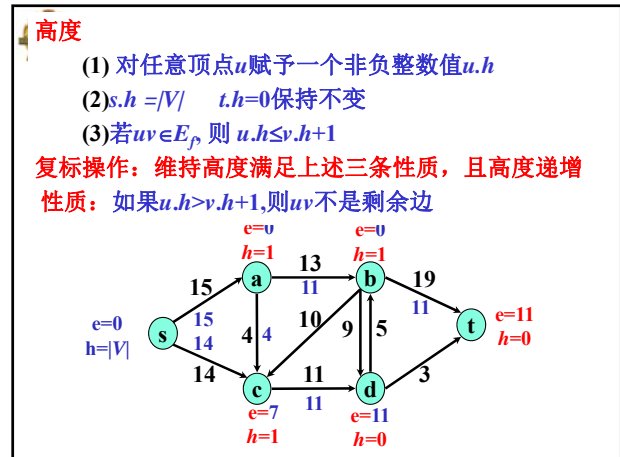
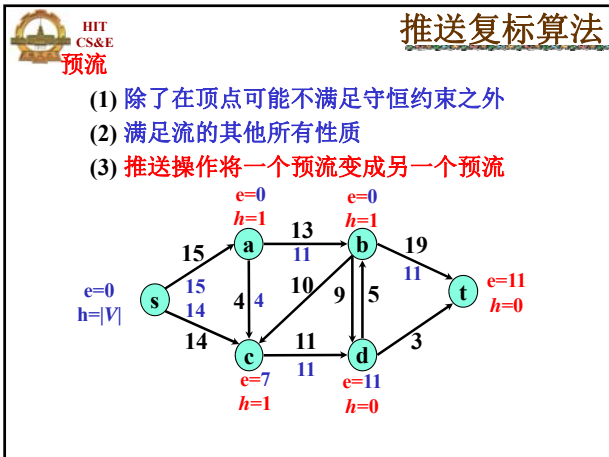
HIT CS&E

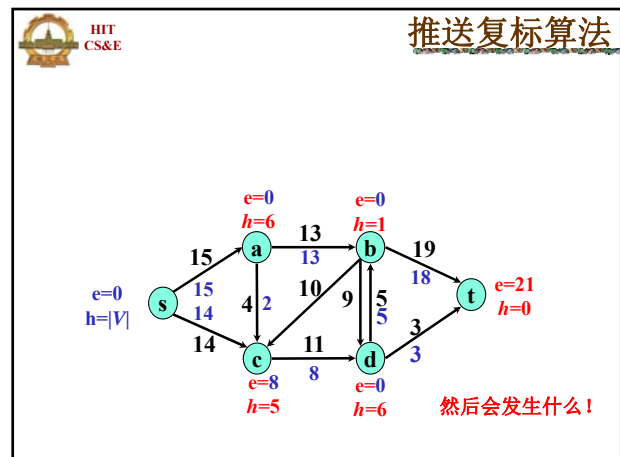
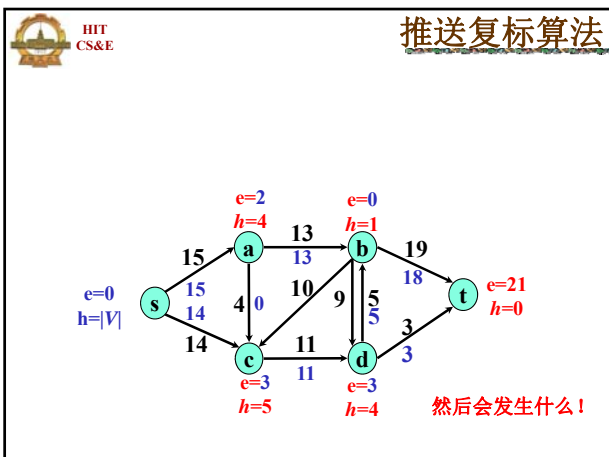
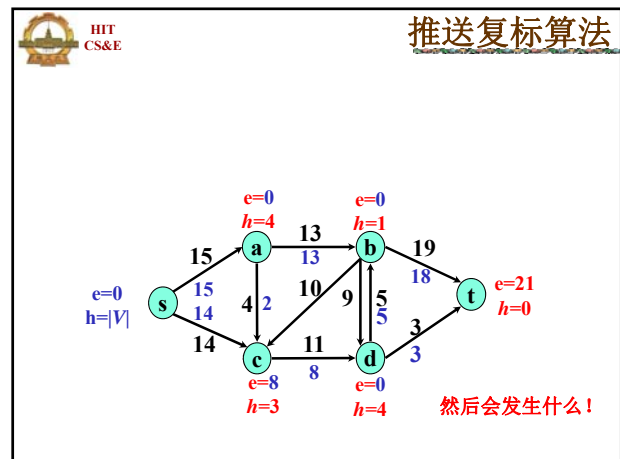
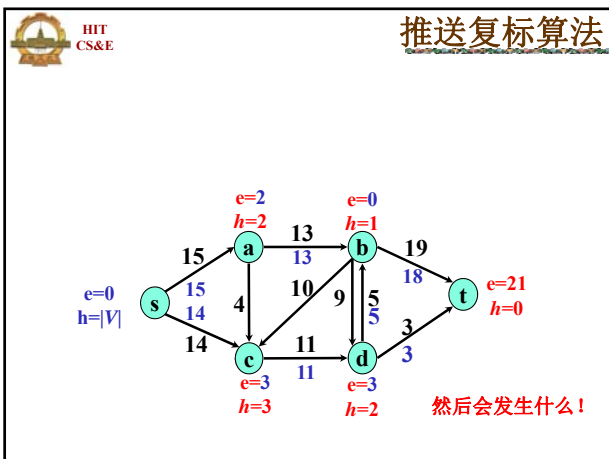
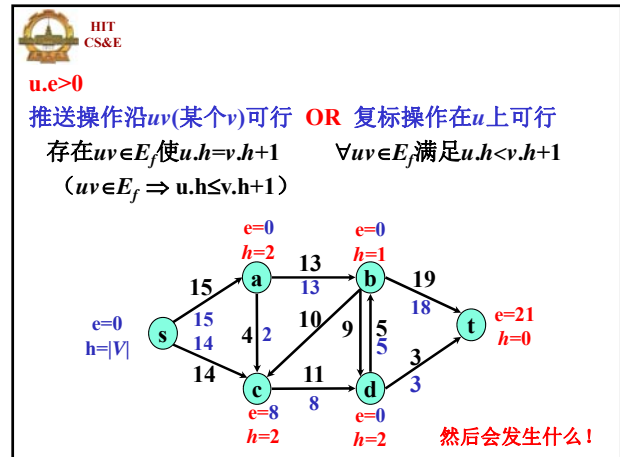
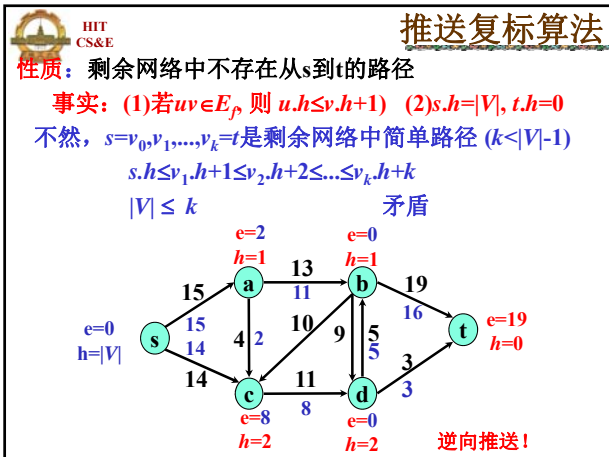
7.1.4 推送复标算法

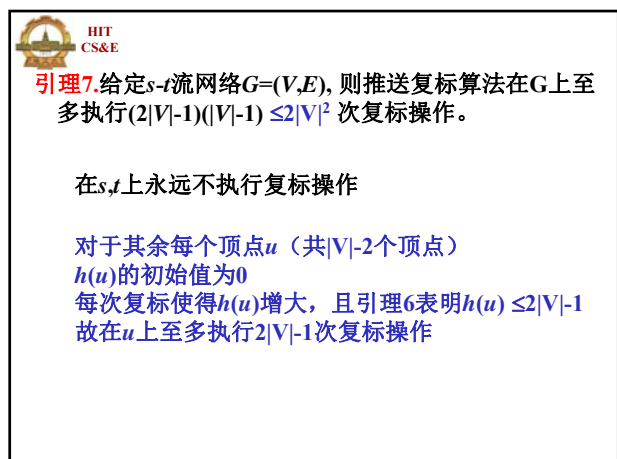
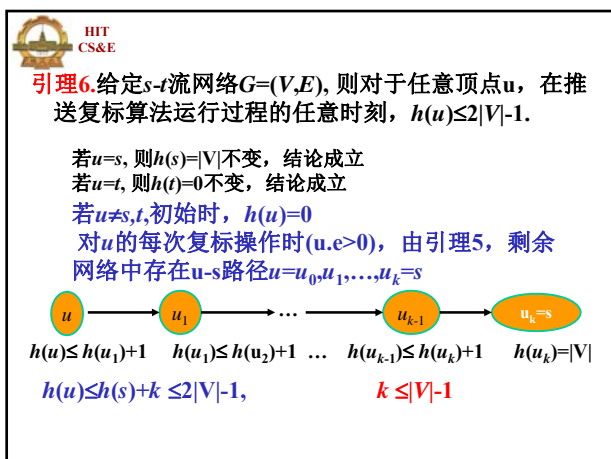
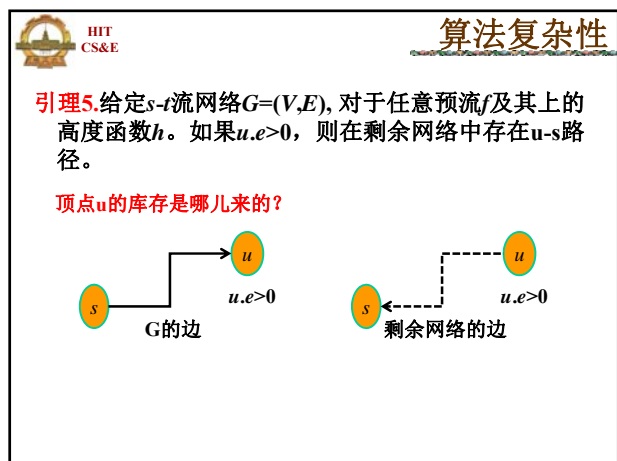
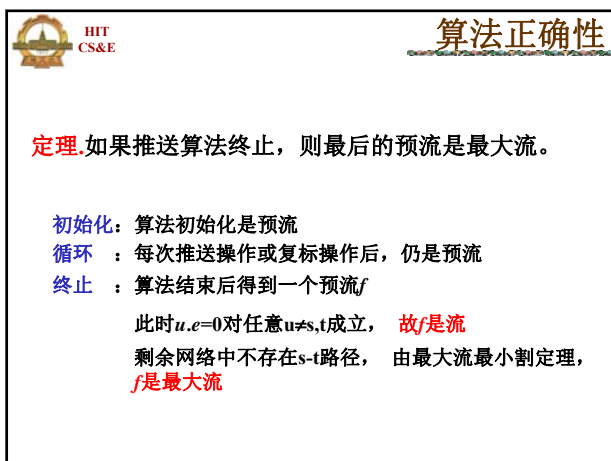
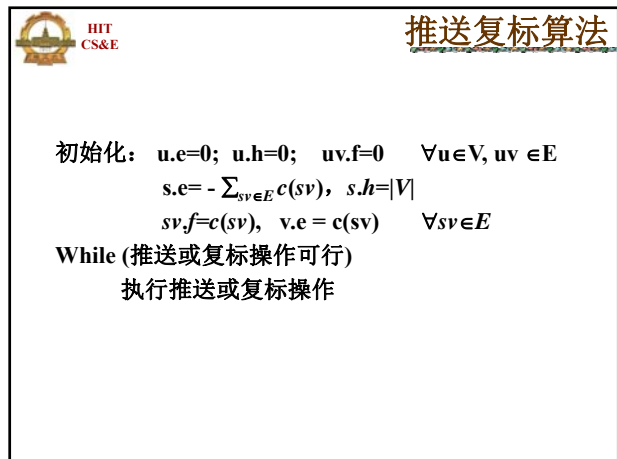
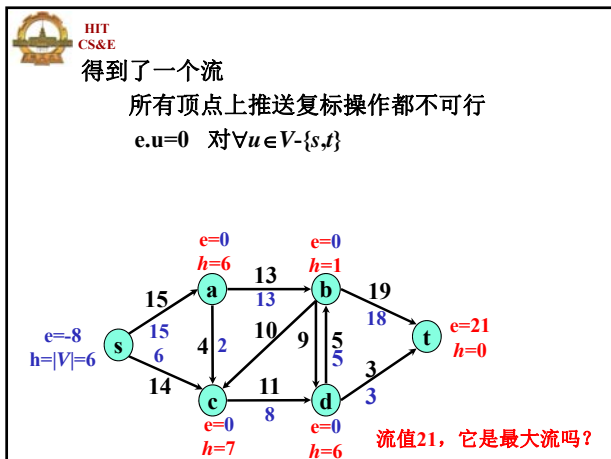
- Ford-Fulkson算法
 - ✓ 构造剩余网络
 - ✓ 选取s-t增广路径来增大流值
 - ✓ 具有全局优化的观点
- 能否只考虑顶点的局部情况
 - ✓ 逐个顶点查看
 - ✓ 仅查看其邻接点
 - ✓ 确定流值的变化









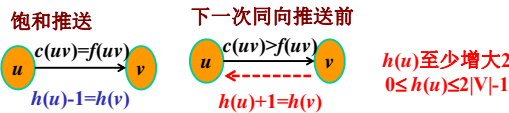




引理8. 给定 s - t 流网络 $G=(V,E)$, 则推送复标算法在 G 上至多执行 $2|V||E|$ 次饱和推送操作。

$\forall u,v \in V$, 考察 u 和 v 之间饱和推送的总次数

- 从 u 到 v 的饱和推送
- 从 v 到 u 的饱和推送
- $uv \in E$ 或 $vu \in E$ 顶点对的个数 $\leq |E|$



- 从 u 到 v 的饱和推送至多 $|V|$ 次
- 同理, 从 v 到 u 的饱和推送至多 $|V|$ 次

引理9. 给定 s - t 流网络 $G=(V,E)$, 则推送复标算法在 G 上至多执行 $4|V|^2(|V|+|E|)$ 次非饱和推送操作。

定义 $\phi = \sum_{u,e>0} h(u)$ 初始时, $\phi=0$ 且 $\phi \geq 0$ 恒成立

复标 顶点 u 使得其高度增加, 导致 ϕ 增大
顶点 u 上的所有复标操作, 导致 ϕ 增大总量 $\leq 2|V|-1$
所有复标操作导致 ϕ 的总增量 $\leq 2|V|^2$

饱和推送
推送前: $u.e>0, v.e=0$ 或 $v.e>0$
推送后: $u.e=0$ 或 $u.e>0, v.e>0$
每次饱和推送导致 ϕ 的增量 $\leq 2|V|$
所有饱和推送导致 ϕ 的增量 $\leq 2|V| \cdot 2|V||E|$

非饱和推送
推送前: $u.e>0, v.e=0$ 或 $v.e>0$
推送后: $u.e=0, v.e>0$
每次非饱和推送导致 ϕ 至少减小 1

由于 $\phi \geq 0$ 恒成立, 总增量 \geq 总减量



定理10. 给定 s - t 流网络 $G=(V,E)$, 则推送复标算法在 G 上至多执行 $O(|V|^2|E|)$ 次基本操作后终止。

由引理7, 算法至多执行 $2|V|^2$ 次复标操作
由引理8, 算法至多执行 $2|V||E|$ 次饱和推送操作
由引理9, 算法至多执行 $4|V|^2(|V|+|E|)$ 次非饱和推送操作



7.1.5 前置复标算法

推送复标算法

- 以不确定的顺序选择推送、复标顶点
- 对每个顶点的处理不彻底
 - ✓ 对 u 的推送或复标操作后, $u.e>0$
 - ✓ 可能转而处理其他顶点
- 时间复杂性 $O(V^2E)$

提高算法性能的着手点

- 如果精细选择推送、复标操作顺序
- 对每个顶点进行彻底处理, 处理后 $u.e=0$



DisCharge操作

彻底处理顶点 u

- $u.e>0$, 则要么推送操作可行, 要么复标操作可行
- 重复在 u 顶点处进行推送或复标操作, 直到 $u.e=0$
- 仅需考察 u 的相邻顶点
- 对 u 维护邻接链表 $L(u)$ $v \in L(u) \Leftrightarrow uv \in E$ 或 $vu \in E$

DisCharge(u)

While $u.e>0$

$v \leftarrow L(u).current$ /*考察当前处理的顶点*/

If $v = \text{Null}$ Then ReLabel(u), $L(u).current = L(u).head$

ElseIf $c(uv)-f(uv)>0$ 且 $u.h=v.h+1$ Then Push(u)

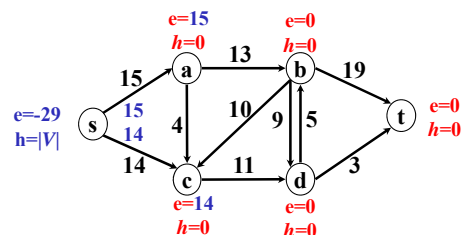
Else $L(u).current \leftarrow L(u).next$

考虑初始化操作之后 DisCharge(c) 的执行过程

$L(c)$:

s	a	b	d	Null
---	---	---	---	------

$v=s \neq \text{Null}$, 不执行复标操作
 $c(cs)-f(cs) = 0 - (-14) = 14$ 但 $c.h \neq s.h+1$ 不执行 Push

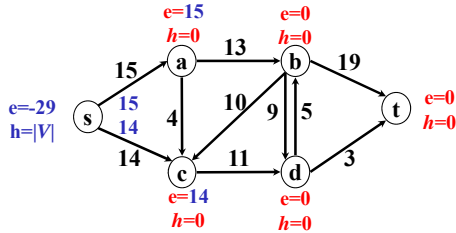


考虑初始化操作之后DisCharge(c)的执行过程

L(c):

s	a	b	d	Null
---	---	---	---	------

$v=a \neq \text{Null}$, 不执行复标操作
 $c(ca)-f(ca) = 0-0 = 0$ 不执行推送操作

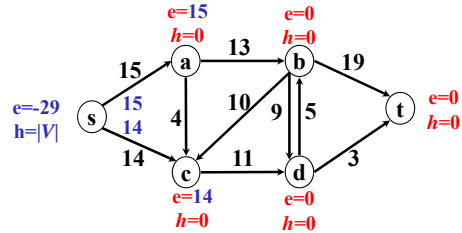


考虑初始化操作之后DisCharge(c)的执行过程

L(c):

s	a	b	d	Null
---	---	---	---	------

$v=b \neq \text{Null}$, 不执行复标操作
 $c(cb)-f(cb) = 0-0 = 0$ 不执行推送操作

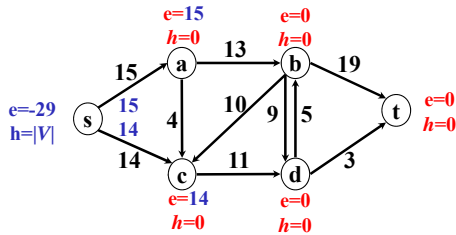


考虑初始化操作之后DisCharge(c)的执行过程

L(c):

s	a	b	d	Null
---	---	---	---	------

$v=d \neq \text{Null}$, 不执行复标操作
 $c(cd)-f(cd) = 11$ 但 $c.h \neq d.h+1$ 不执行推送操作

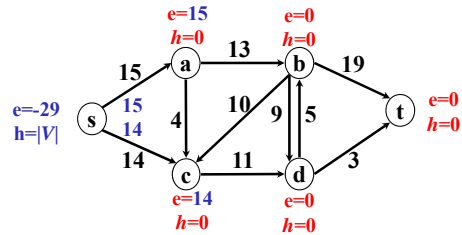


考虑初始化操作之后DisCharge(c)的执行过程

L(c):

s	a	b	d	Null
---	---	---	---	------

$v=\text{Null}$, 执行复标操作

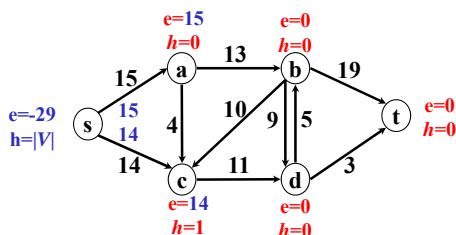


考虑初始化操作之后DisCharge(c)的执行过程

L(c):

s	a	b	d	Null
---	---	---	---	------

$v=\text{Null}$, 执行复标操作

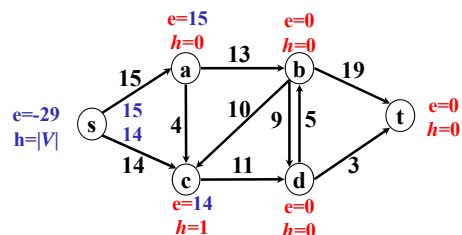


考虑初始化操作之后DisCharge(c)的执行过程

L(c):

s	a	b	d	Null
---	---	---	---	------

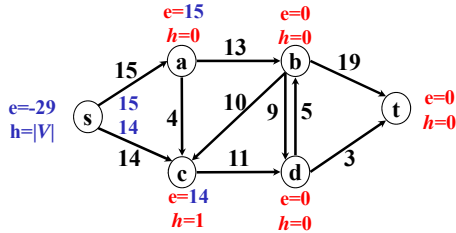
$v=s \neq \text{Null}$, 不执行复标操作
 $c(cs)-f(cs) = 0-(-14) = 14$ 但 $c.h \neq s.h+1$ 不执行Push



考虑初始化操作之后DisCharge(c)的执行过程

$L(c): [s, a, b, d, \text{Null}]$

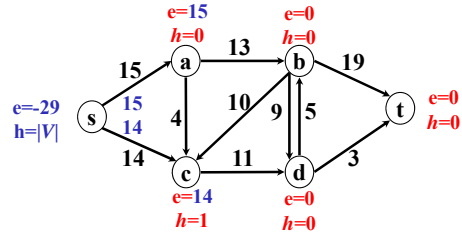
$v=a \neq \text{Null}$, 不执行复标操作
 $c(ca)-f(ca) = 0-0 = 0$ 不执行推送操作



考虑初始化操作之后DisCharge(c)的执行过程

$L(c): [s, a, b, d, \text{Null}]$

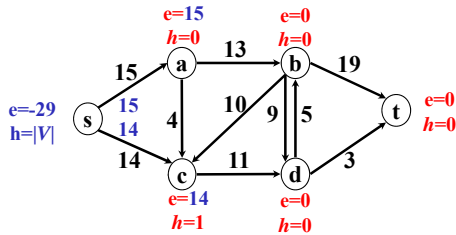
$v=b \neq \text{Null}$, 不执行复标操作
 $c(cb)-f(cb) = 0-0 = 0$ 不执行推送操作



考虑初始化操作之后DisCharge(c)的执行过程

$L(c): [s, a, b, d, \text{Null}]$

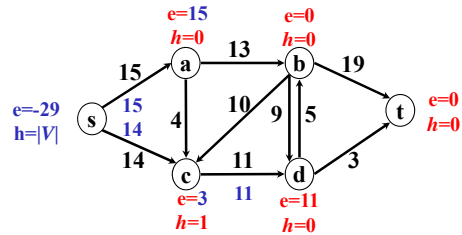
$v=d \neq \text{Null}$, 不执行复标操作
 $c(cd)-f(cd) = 11$ 且 $c.h = d.h+1$ 执行Push操作



考虑初始化操作之后DisCharge(c)的执行过程

$L(c): [s, a, b, d, \text{Null}]$

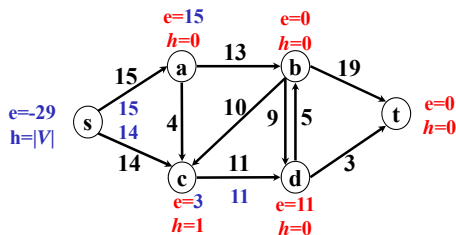
$v=d \neq \text{Null}$, 不执行复标操作
 $c(cd)-f(cd) = 11$ 且 $c.h = d.h+1$ 执行Push操作



考虑初始化操作之后DisCharge(c)的执行过程

$L(c): [s, a, b, d, \text{Null}]$

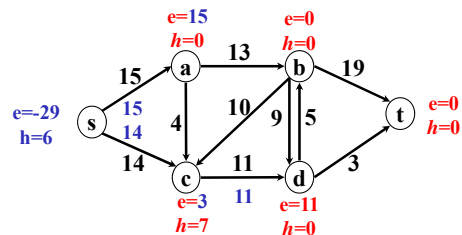
$v=\text{Null}$, 执行复标操作
 $c.h = 1 + \min \{v.h \mid c(cv)-f(cv) > 0\} = |V|+1$



考虑初始化操作之后DisCharge(c)的执行过程

$L(c): [s, a, b, d, \text{Null}]$

$v=\text{Null}$, 执行复标操作
 $c.h = 1 + \min \{v.h \mid c(cv)-f(cv) > 0\} = |V|+1$

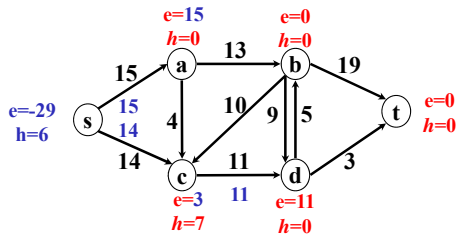


考虑初始化操作之后DisCharge(c)的执行过程

L(c):

s	a	b	d	Null
---	---	---	---	------

$v=s \neq \text{Null}$, 不执行复标操作
 $c(cs)-f(cs)=14$ 且 $c.h = c.s+1$ 执行Push

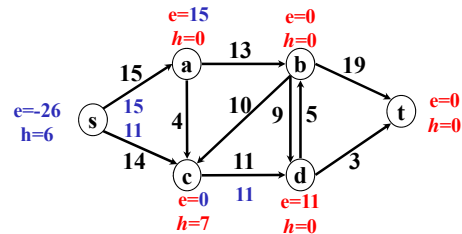


考虑初始化操作之后DisCharge(c)的执行过程

L(c):

s	a	b	d	Null
---	---	---	---	------

$v=s \neq \text{Null}$, 不执行复标操作
 $c(cs)-f(cs)=14$ 且 $c.h = c.s+1$ 执行Push



HIT CS&E 前置复标算法

顶点链表B: $u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_{n-2} \rightarrow \text{Null}$

邻接链表L: $L(u_1), L(u_2), \dots, L(u_{n-2})$

依次处理B中的每个顶点u

- 处理过程即调用DisCharge操作使得 $u.e=0$
- 处理后将u前置于B的开始位置
- 然后顺序处理下一个顶点

到达B的结束位置, 则 $u.e=0$ 对所有顶点成立
 即得到最大流

$u_1.e=0, u_2.e=0, \dots, u_{n-2}.e=0$

$u_1 \rightarrow u_2 \rightarrow \dots \rightarrow u_{n-2} \rightarrow \text{Null}$

$L(u_1), L(u_2), \dots, L(u_{n-2})$

HIT CS&E 前置复标算法

初始化: $u.e=0; u.h=0; uv.f=0 \quad \forall u \in V, uv \in E$
 $s.e = -\sum_{sv \in E} c(sv), s.h = |V|$
 $sv.f = c(sv), v.e = c(sv) \quad \forall sv \in E$
 创建链表B管理V-{s,t}的所有顶点
 $L(v) \leftarrow v$ 的相邻顶点链表
 $L(v).current \leftarrow L(v).head$

1. $u \leftarrow B.head$
2. While $u \neq \text{Null}$
3. $oldHeight \leftarrow u.h$
4. DisCharge(u);
5. If $u.h > oldHeight$ then 将u前置到B的前端
6. $u \leftarrow u.next$

HIT CS&E 前置复标算法分析

引理1. 前置复标算法终止后, 则最后的预流是最大流。

初始化: 算法初始化是预流

循环: 每次推送操作或复标操作后, 仍是预流

终止: 算法结束后得到一个预流f

算法结束后 $u.e=0$ 对任意 $u \neq s, t$ 成立, 故f是流

类似与推送复标算法, 可以证明剩余网络中不存在s-t路径, 由最大流最小割定理, f是最大流

HIT CS&E 前置复标算法分析

引理2. 前置复标算法在 $O(V^3)$ 个基本操作之后必然终止。

前置复标算法是推送复标操作的特例

- 至多执行 $2V^2$ 个复标操作
- 至多执行 $2VE$ 个饱和推送操作
- 只需限定非饱和推送的个数

非饱和推送操作 个数 \leq DisCharge执行次数
 每次非饱和推送执行后, $u.e=0$, DisCharge操作结束

考察执行复标操作的两个DisCharge操作之间

- 算法不改变任意顶点的高度, 故算法顺序扫描B中顶点
- 算法顺序处理链表B中的一个连续区段
- 该区段的长度不超过B的总长度V-2

非饱和推送至多执行 $2V^2 \cdot (V-2) < 2V^3$ 个

HIT CS&E

7.2 匹配算法

7.2.1 匹配与覆盖
7.2.2 最大二分匹配算法
7.2.3 最大权值二分匹配

HIT CS&E

7.2.1 覆盖与匹配

匹配——图 $G=(V,E)$ 中没有公共端点的一组边 M

- ◆ 匹配边—— M 中的边
- ◆ 自由边—— E/M 中的边
- ◆ 被浸润的顶点—— M 中边的端点
- ◆ 未被浸润的顶点——其他顶点

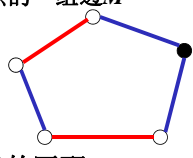
完美匹配——浸润 G 的每个顶点的匹配

最大匹配——边的条数达到最大值的匹配

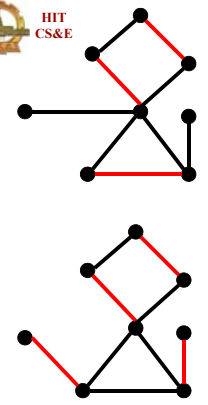
性质：完美匹配是最大匹配，反之不然

最大匹配问题

输入：图 $G=(V,E)$
输出： G 的最大匹配 M



HIT CS&E

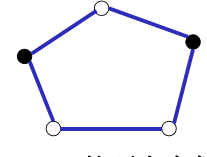


最大匹配
非完美匹配

最大匹配
完美匹配

HIT CS&E

顶点覆盖——图 $G=(V,E)$ 中的一个顶点子集 C
 E 中每条边都至少有一个端点在 C 中



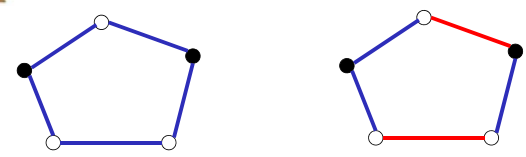
最小顶点覆盖—— G 的顶点个数最少的覆盖

最小顶点覆盖问题

输入：图 $G=(V,E)$
输出： G 的最小顶点覆盖

HIT CS&E

弱对偶性



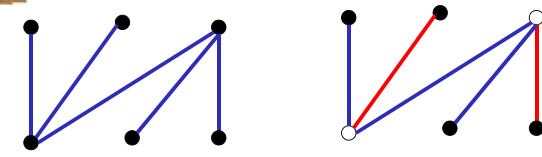
若 C 是图 G 的任意顶点覆盖， M 是图 G 的任意匹配

- ◆ M 中每条边都至少有一个端点在 C 中
- ◆ M 中任意两条边不存在公共端点

故 $|M| \leq |C|$

定理：图 G 的最小顶点覆盖 C 和最大匹配 M 满足 $|M| \leq |C|$
在二分图 G 中， $|M| = |C|$

HIT CS&E



若 C 是图 G 的任意顶点覆盖， M 是图 G 的任意匹配

- ◆ M 中每条边都至少有一个端点在 C 中
- ◆ M 中任意两条边不存在公共端点

故 $|M| \leq |C|$

定理：图 G 的最小顶点覆盖 C 和最大匹配 M 满足 $|M| \leq |C|$
在二分图 G 中， $|M| = |C|$

HIT CS&E

研究匹配算法的意义

工作分配

输入: n 个人 p_1, \dots, p_n , n 项工作 J_1, \dots, J_n , 第 i 个人胜任其中 k 项工作
 输出: 是否存在工作分配方案使得每个人完成1项自己胜任的工作

计算节点 实习单位
 计算任务 实习人员

目前, 大规模图数据管理已经非常盛行
 很多高效算法都以匹配算法或覆盖算法为基础!

HIT CS&E

7.2.2 最大二分匹配算法

定理: 图 G 的最小顶点覆盖 C 和最大匹配 M 满足 $|M| \leq |C|$
 在二分图 G 中, $|M| = |C|$

这意味着二分图上的最大匹配可以这样求解

- ◆ 初始化一个匹配 M
- ◆ 不断地增大 M
- ◆ M 无法增大时, 找出一个顶点覆盖 C 使得 $|M| = |C|$
- ◆ M 是最大匹配, C 是最小覆盖

HIT CS&E

初始化

HIT CS&E

增大

M 交错路径: 边交替出现在 M 和 $E-M$ 的路径
 M 增广路径: 端点未被 M 浸润的交错路径

找出增广路径, 就能增大 M ! 怎么找呢?

HIT CS&E

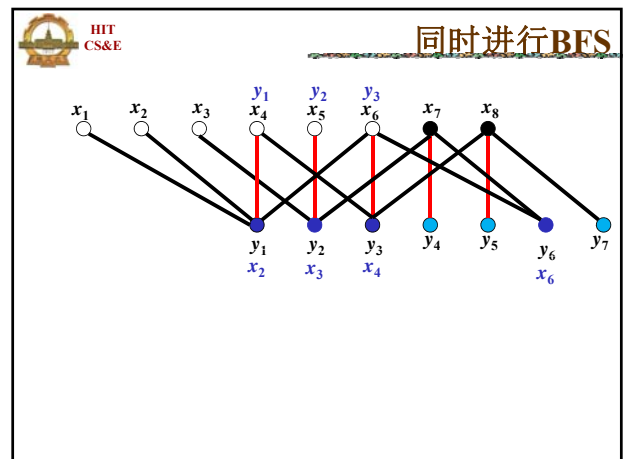
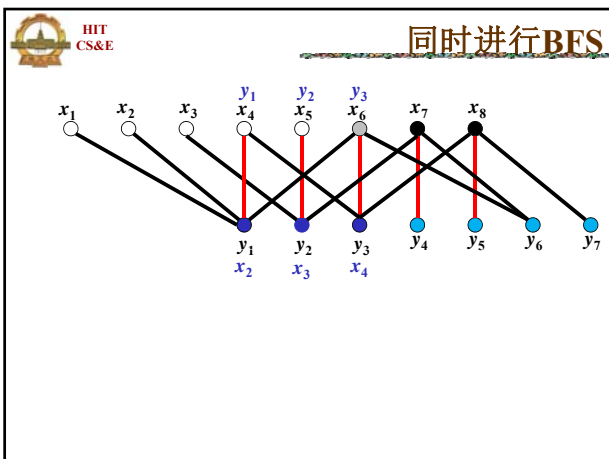
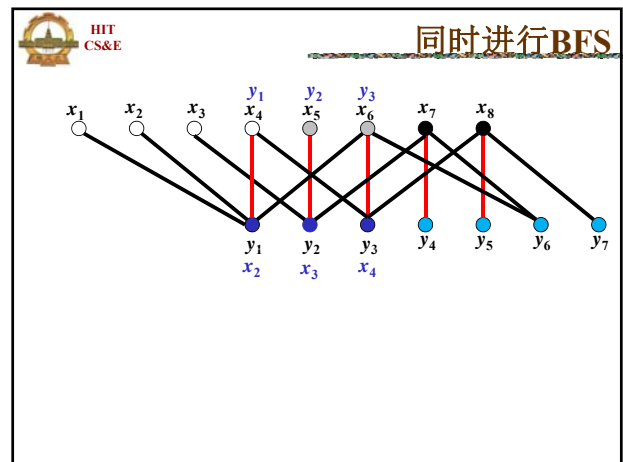
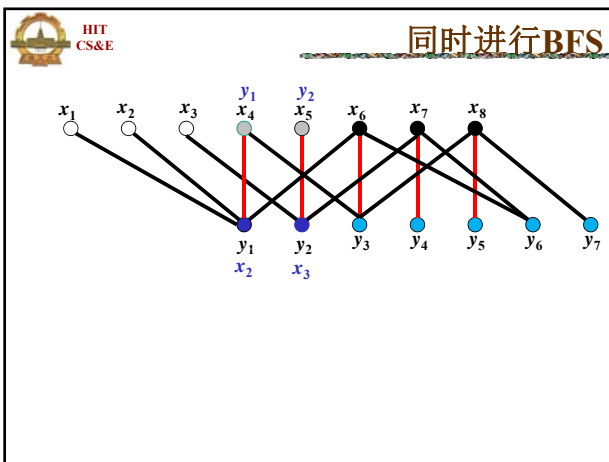
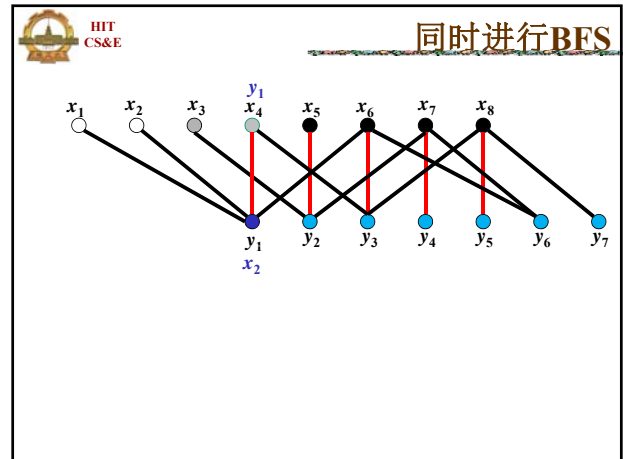
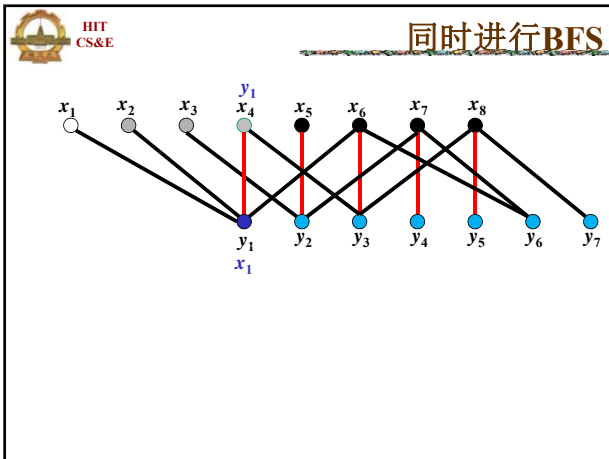
通过BFS找增广路径

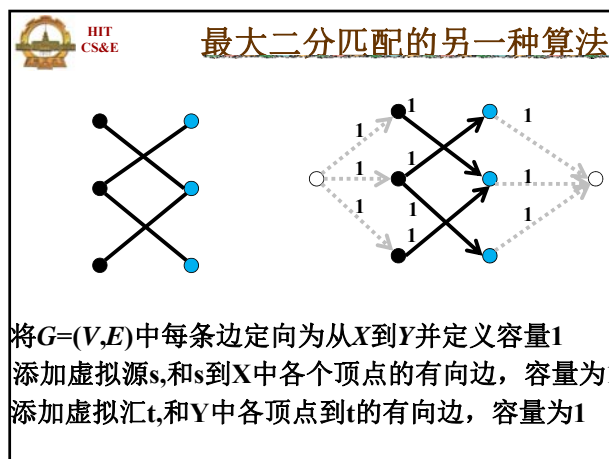
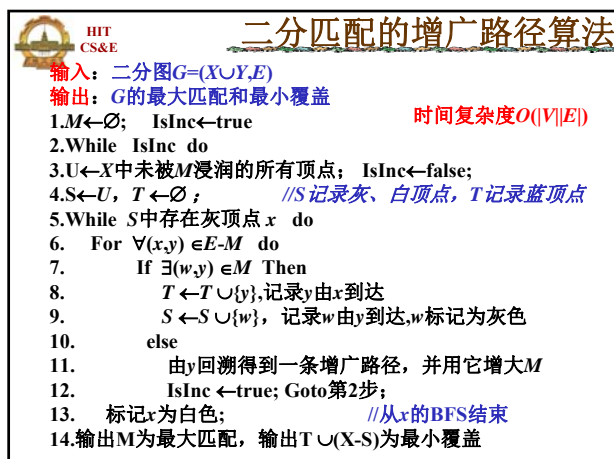
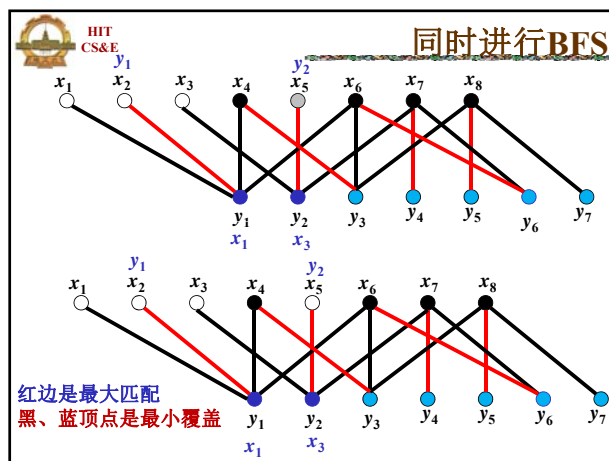
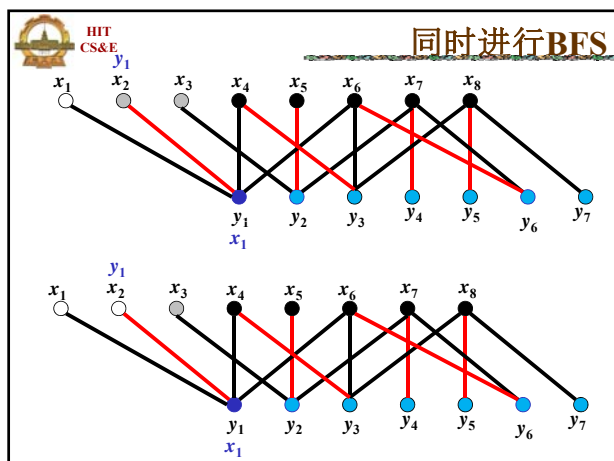
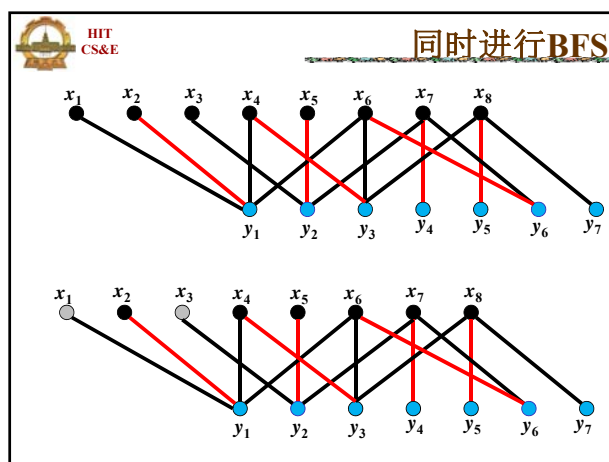
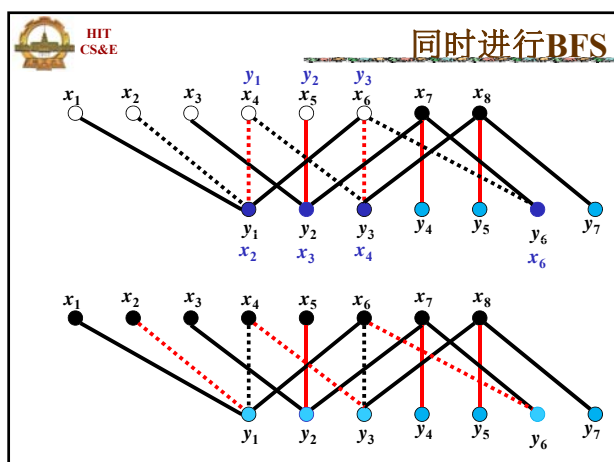
1. $U \leftarrow X$ 中未被 M 浸润的所有顶点
2. For $\forall x \in U$ do
 如果存在从 x 出发的 M 增广路径, 则增大 M
 Goto 1
3. M 是最大匹配

能够以更高效的方式进行广度优先搜索呢?

HIT CS&E

同时进行BFS





HIT CS&E **基于最大流的最大二分匹配算法**

f 是新图上的最大流
当且仅当
 E 中流量大于0的边构成 G 的最大匹配 M

HIT CS&E **7.2.3最大权值二分匹配**

计算结点
◆边上的权值表示计算效率
◆最大匹配就是计算效率最高的计算任务分配方案
计算任务

匹配 M 的权值 $c(M)$: M 中各条边的权值之和

最大权值二分匹配问题
输入: 加权二分图 $G=(X \cup Y, E), w: E \rightarrow R$
输出: G 上的最大权值二分匹配 M

HIT CS&E

最大权值二分匹配问题
输入: 加权完全二分图 $G=(X \cup Y, E), w: E \rightarrow R$
输出: G 上的最大权值二分匹配 M

HIT CS&E

最大权值二分匹配问题
输入: 加权完全二分图 $K_{n,n}, w: E \rightarrow R$
输出: G 上的最大权值二分匹配 M

HIT CS&E

最大权值二分匹配问题
输入: 加权完全二分图 $K_{n,n}, w: E \rightarrow R$
输出: G 上的最大权值二分匹配 M

HIT CS&E **最大权值顶点覆盖**

加权 $K_{n,n}$ 的顶点覆盖
◆两个 n 维实值向量 u, v
◆ $u_i + v_j \geq w_{ij}$

最小权值顶点覆盖问题
输入: 加权二分图 $K_{n,n}, w: E \rightarrow R$
输出: 分量之和达到最小值的顶点覆盖

对偶关系

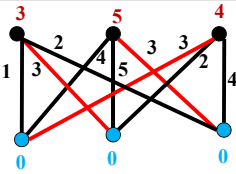
任意完美匹配 M , 任意覆盖 u, v

$$u_i + v_j \geq w_{ij}$$

$$\sum u_i + \sum v_j \geq \sum w_{ij}$$

$$c(u, v) \geq c(M)$$

定理: 在加权 $K_{n,n}$ 上, 任意完美匹配 M 和任意顶点覆盖 u, v 比满足 $c(M) \leq c(u, v)$, 并且 M 是最大权值匹配当且仅当 $c(M) = c(u, v)$, 此时 M 中的每条边 ij 均满足 $u_i + v_j = w_{ij}$



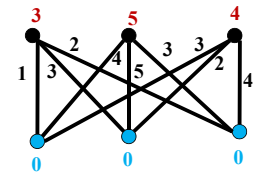
对偶关系给出了问题的求解思路

初始化顶点覆盖 u, v

判断 $\{ij: u_i + v_j = w_{ij}\}$ 中能否找出完美匹配 M

若是, 则 M 是最大匹配, u, v 是最小覆盖

否则, 调整 u, v



$u_i \leftarrow \max\{w_{ij}\}$
 $v_j \leftarrow 0$

初始化

相等子图
仅含 $u_i + v_j = w_{ij}$ 的所有边

$u_i \leftarrow \max\{w_{ij}\}$
 $v_j \leftarrow 0$

初始化

红边: 最大匹配
蓝、白顶点: 最小覆盖
调整端点不在最小覆盖中的边, 让它出现在相等子图中

$2=3-1$ $4=5-1$ $1=0+1$

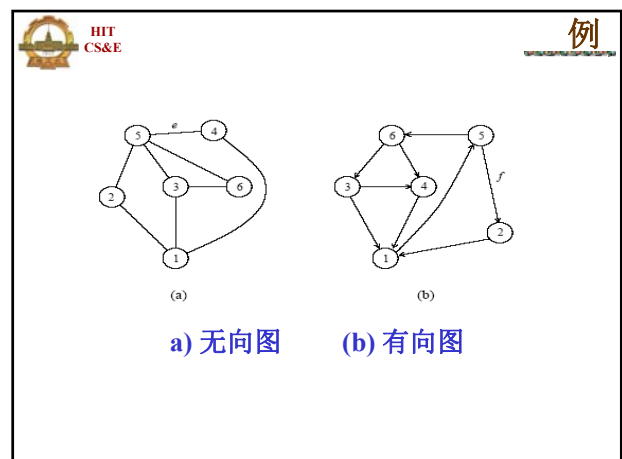
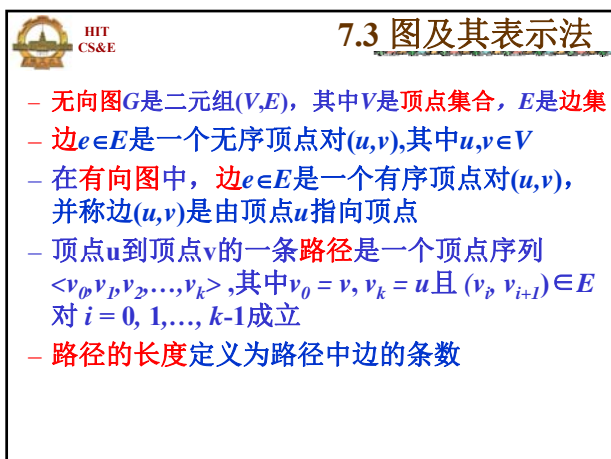
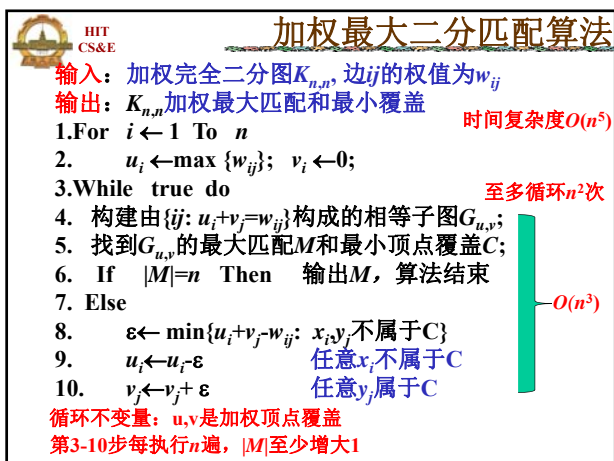
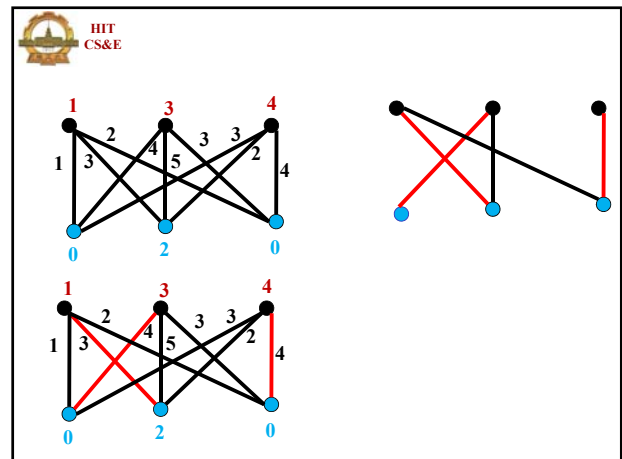
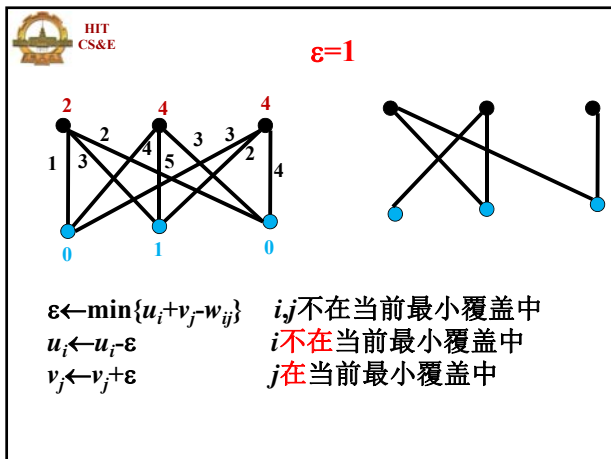
调整覆盖 u, v

$\varepsilon \leftarrow \min\{u_i + v_j - w_{ij}\}$
 $u_i \leftarrow u_i - \varepsilon$
 $v_j \leftarrow v_j + \varepsilon$

调整覆盖 u, v

$\varepsilon=1$

i, j 不在当前最小覆盖中
 i 不在当前最小覆盖中
 j 在当前最小覆盖中



HIT
CS&E

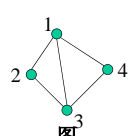
- 无向图是**连通**的，如果其任意两个顶点之间均有一条路径
- 森林**指的是无环图，**树**是连通的森林。
- 如果图的每条边均有一个权值与之关联，则称该图为**加权图**。

HIT
CS&E

图的表示

给定图 $G = (V, E)$ 如何在计算机内存储它?

- 邻接矩阵**: 顶点 i 表示为矩阵的第 i 行和第 i 列; 矩阵第 i 行第 j 列的元素被置为1(或者相应边的权值), 如果 $(i, j) \in E$; 否则置为0
- 邻接表**: 顶点作为数组的索引, 每个顶点用一个链表存储与之邻接的所有顶点



图

	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

图的邻接矩阵

	1	2	3	4
1	→	2	3	4
2	→	1	3	
3	→	1	2	4
4	→	1	3	

图的邻接表

HIT
CS&E

表示方法的比较

给定图 $G = (V, E)$, $|V|=n, |E|=m$?

- 存储空间**
邻接矩阵: $O(n^2)$ 邻接表: $O(m+n)$
- 查看给定顶点的所有相邻顶点**
邻接矩阵: $O(n)$ 邻接表: $O(m/n)$
- 检查给定的顶点对是否是 G 的一条边**
邻接矩阵: $O(1)$ 邻接表: $O(m/n)$

—具体采用何种数据结构, 看具体的任务主要涉及何种操作
—稀疏图往往采用邻接表存储

HIT
CS&E

7.4 基本图论算法

- 7.2.1 广度优先搜索
- 7.2.2 深度优先搜索
- 7.2.3 拓扑排序
- 7.2.4 连通分支

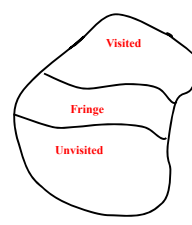
HIT
CS&E

搜索算法

- 访问图中边访问所有顶点的系统化方法
- 可以发现图的很多结构信息
- 很多其他图论算法均是这些基本搜索算法的精细化结果
- 图的搜索算法是图论算法的核心

HIT
CS&E

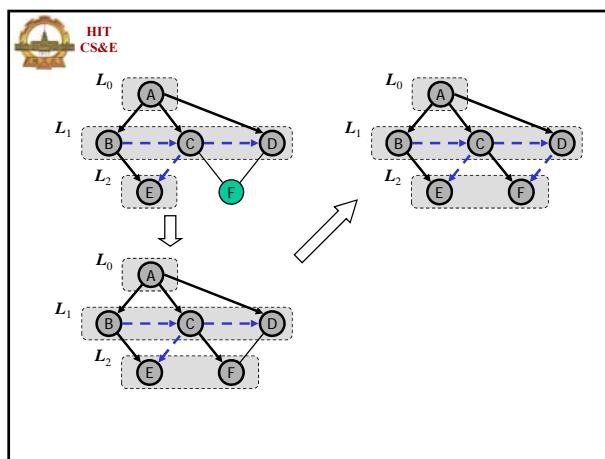
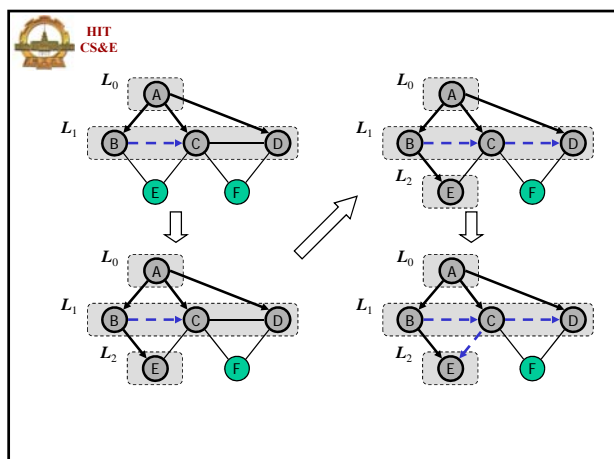
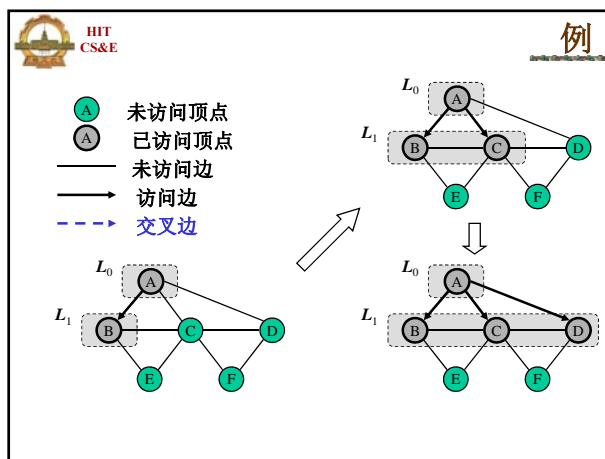
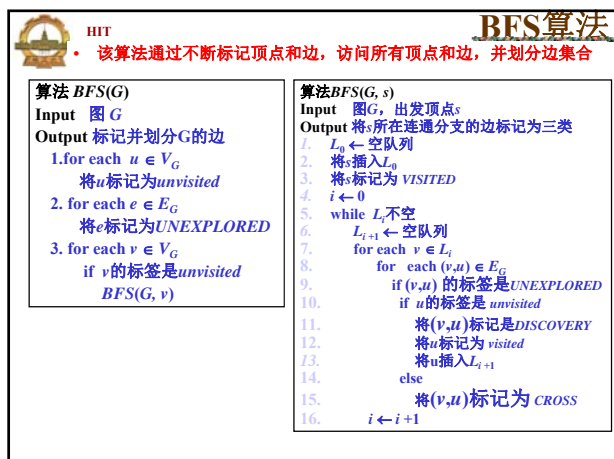
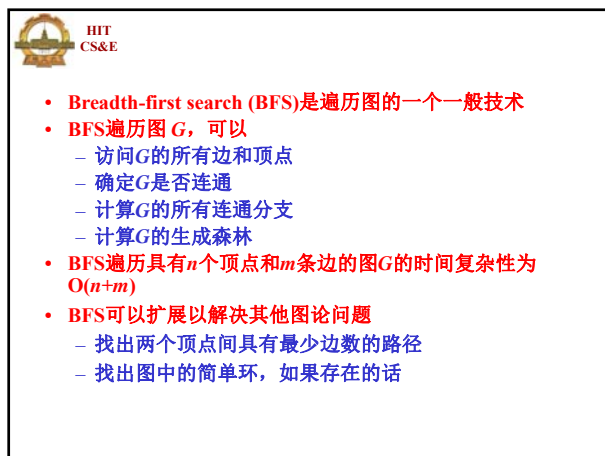
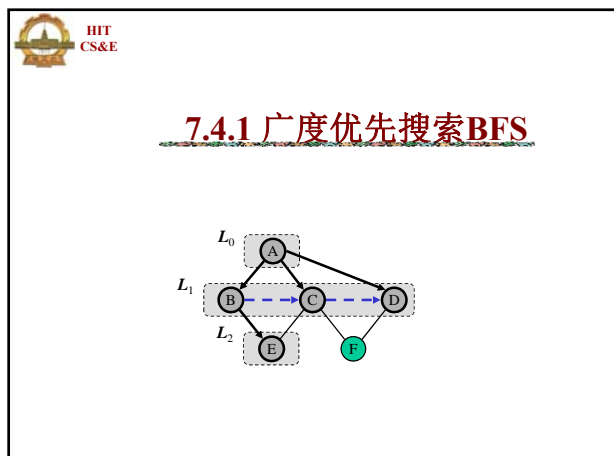
图搜索算法中的三类顶点



搜索过程中, 顶点集可以看成如下3个部分的并集

- **已访问顶点**: 已被算法处理过的顶点。
- **边缘顶点**: 马上将被访问的顶点。
- **未访问顶点**: 仍未被算法处理过的顶点

❖ DFS算法用**栈**存储边缘顶点, 而BFS算法用**队列**存储边缘顶点



HIT
记号

BFS算法的性质

G_s : 顶点 s 所在的连通分支

Property 1
 $BFS(G, s)$ 访问 G_s 中的所有边和顶点

Property 2
 $BFS(G, s)$ 找出的所有发现边构成 G_s 的一棵生成树 T_s

Property 3
对任意 $v \in L_i$

- 在 T_s 中, s 到 v 的路径有 i 条边
- 在 G_s 中, s 到 v 的任意路径至少有 i 条边

HIT
CS&E

BFS算法分析

- 标记和获取顶点或边的时间开销为 $O(1)$
- 每个顶点被标记两次
 - 一次将其标记为 **unvisited**
 - 一次将其标记为 **visited**
- 每条边被标记两次
 - 一次将其标记为 **UNEXPLORED**
 - 一次将其标记为 **DISCOVERY** 或 **CROSS**
- 每个顶点仅被唯一地插入某个 L_i
- 查找顶点的邻边仅被每个顶点调用一次 **第8步**
- 只要图是以邻接表存储的, BFS的运行时间为 $O(n + m)$
 - 注意 $\sum_v \deg(v) = 2m$

HIT
CS&E

7.4.2 深度优先搜索DFS

HIT
CS&E

- **Depth-first search (DFS)**是遍历图的一个一般技术
- DFS遍历图 G , 可以
 - 访问 G 的所有边和顶点
 - 确定 G 是否连通
 - 计算 G 的所有连通分支
 - 计算 G 的生成森林
- DFS遍历具有 n 个顶点和 m 条边的图 G 的时间复杂性为 $O(n+m)$
- DFS可以扩展以解决其他图论问题
 - 找出两个顶点间的路径
 - 找出图中的简单环, 如果存在的话

HIT

• 该算法通过不断标记顶点和边, 访问所有顶点和边, 并划分边集合

算法 $DFS(G)$
Input 图 G
Output 标记并划分 G 的边

1. for each $u \in V_G$
 将 u 标记为 **unvisited**
2. for each $e \in E_G$
 将 e 标记为 **UNEXPLORED**
3. for each $v \in V_G$
 if v 的标签是 **unvisited**
 $Recursive_DFS(G, v)$

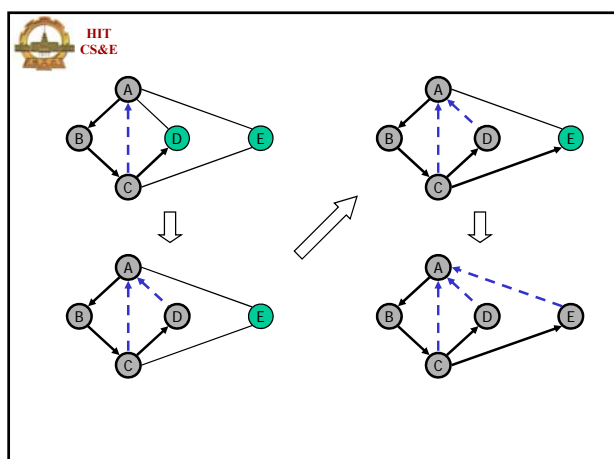
算法 $Recursive_DFS(G, s)$
Input 图 G , 出发顶点 s
Output 将 s 所在连通分支的边标记为两类

1. 将 s 标记为 **VISITED**
2. for each $(s, u) \in E_G$
 if (s, u) 的标签是 **UNEXPLORED**
 if u 的标签是 **unvisited**
 将 (s, u) 标记为 **DISCOVERY**
 将 u 标记为 **visited**
 $Recursive_DFS(G, u)$
 else
 将 (s, u) 标记为 **Back**

很容易将上述算法改为栈的形式

HIT
CS&E

例



HIT CS&E

DFS与迷宫

- DFS算法很像走迷宫的经典策略
 - We mark each intersection, corner and dead end (vertex) visited
 - We mark each corridor (edge) traversed
 - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)

HIT CS&E

DFS算法的性质

Property 1
 $DFS(G, v)$ 访问 v 所在连通分支的所有顶点和边

Property 2
 $DFS(G, v)$ 找出的所有发现边构成 v 所在连通分支的生成树

Property 3
 有向图 G 是一个无环图当且仅当 G 上的 DFS 过程没有回边

HIT CS&E

DFS算法分析

- 标记和获取顶点或边的时间开销为 $O(1)$
- 每个顶点被标记两次
 - 一次将其标记为 **unvisited**
 - 一次将其标记为 **visited**
- 每条边被标记两次
 - 一次将其标记为 **UNEXPLORED**
 - 一次将其标记为 **DISCOVERY** 或 **BACK**
- 查找顶点的邻边仅被每个顶点调用一次 **第2步**
- 只要图是以邻接表存储的, BFS 的运行时间为 $O(n + m)$
 - 注意 $\sum \deg(v) = 2m$

HIT CS&E

作业

分别修改BFS和DFS以完成以下任务

- 确定 G 是否连通
- 计算 G 的所有连通分支
- 计算 G 的生成森林
- 找出两个顶点间的路径
- 找出图中的简单环, 如果存在的话

HIT CS&E

7.4.3 拓扑排序

HIT CS&E 偏序集合的拓扑排序

- 拓扑排序
 - 输入: 偏序集合(S, \leq)
 - 输出: S 的拓扑序列是 $\langle s_1, s_2, \dots, s_n \rangle$,
满足: 如果 $s_i \leq s_j$, 则 s_i 排在 s_j 的前面.

- 例

拓扑排序:
 $s_1 s_3 s_7 s_4 s_9 s_5 s_2 s_8 s_6$

HIT CS&E 有向图的拓扑排序

- 拓扑排序
 - 输入: 无环有向图 $G=(V,E)$
 - 输出: G 的所有顶点的一个拓扑排序
即, 如果 $uv \in E$, 则 u 在该排序中位于 v 之前

- 例

拓扑排序:
 $C A B D E G F H I$

HIT CS&E 有向图的拓扑排序算法

算法 TOPOLOGICAL_SORT(G)
Input 无环有向图 $G=(V,E)$
Output G 的所有顶点的拓扑排序

1. 初始化空链表 L
2. 调用DFS算法计算每个顶点 v 的结束时间 $f(v)$
3. 当 v 的结束时间 $f(v)$ 被计算出来时, 将 v 插入 L 的最前端
3. 链表 L 中的顶点顺序即为一个拓扑序

HIT CS&E 例

$m r u y v w z q t x$

HIT CS&E 算法分析

定理 对无环有向图 G , TOPOLOGICAL_SORT(G)
得到 G 的一个拓扑排序

证明: 仅需证明, 如果 $uv \in E$ 则 $f(v) < f(u)$.

考虑 uv 被DFS访问的时刻, v 要么已被访问完 ($f(v)$ 已被计算出来), 要么 v 仍未被访问过, 否则将出现环。

对于第一种情况, 显然有 $f(v) < f(u)$.

对于第二种情况, v 是 u 的后代, DFS算法必然会先结束对 v 的访问, 故 $f(v) < f(u)$.

时间复杂度分析
即DFS的时间复杂度, $O(|V|+|E|)$

HIT CS&E 7.4.4 有向图的强连通分枝分解

- 将有向图分解为强连通分枝是DFS的一个经典应用
- 许多应用需要将有向图分解为强连通分枝, 然后再对每个强连通分枝应用某种操作或算法
- 给定有向图 $G=(V,E)$, G 的一个强连通分枝指的是一个极大子集 $C \subseteq V$ 使得 $\forall u, v \in C$ 均有 $u \leftrightarrow v$.
- 将有向图 $G=(V,E)$ 的所有边反向后得到的图成为 G 的转置, 记为 G^T
 - 若 G 以邻接矩阵 A 给出, G^T 的邻接矩阵即为 A^T
 - 若 G 以邻接表给出, 请给出一个算法计算 G^T 的邻接表

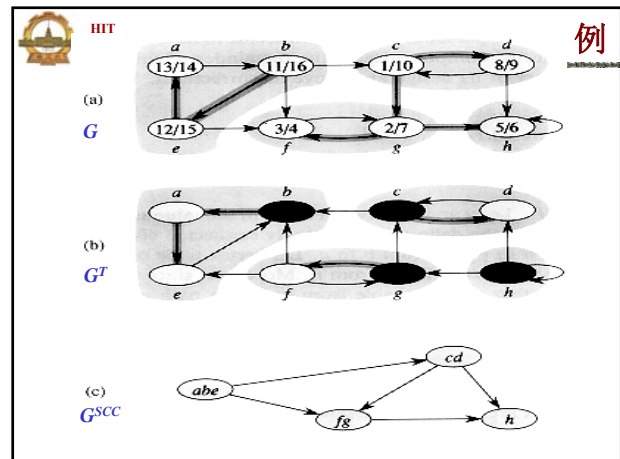
HIT CS&E

注意, G 的转置和 G 具有相同的连通分枝

G 的连通分枝图 $G^{SCC} = (V^{SCC}, E^{SCC})$ 定义如下:

- G 的每个连通分枝对应 V^{SCC} 中的一个顶点
- $x, y \in V^{SCC}$, 如果 x 对应连通分枝到 y 对应的连通分枝在 G 中有一条有向边, 则 $(x, y) \in E^{SCC}$

连通分枝分解算法的分析依赖于 G^{SCC} 的一些性质



HIT CS&E

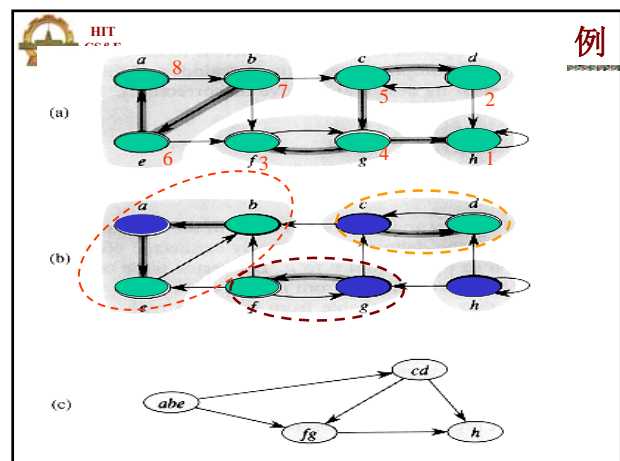
强连通分枝分解算法

算法 **STONGLY_CONNECT_COMPONENTS(G)**

Input 无环有向图 $G=(V,E)$

Output G 的强连通分枝分解

1. 在 G 上调用 DFS 计算每个顶点 v 的结束时间 $f(v)$
2. 计算 G^T
3. do
4. 在 G^T 的剩余顶点中, 从最大 $f(v)$ 对应的顶点开始进行 DFS 访问
5. 如果当前访问顶点没有未被访问过的边, 则输出当前访问过的顶点 作为一个连通分枝
6. while(G^T 仍有顶点未被访问)



HIT CS&E

算法分析

引理1 设 C 和 C' 是有向图 $G=(V,E)$ 中两个不同的连通分枝, 且 $u, v \in C, u', v' \in C'$. 如果 $u \rightarrow u'$ 则 $v' \rightarrow v$ 不成立

HIT CS&E

- 将 DFS 算法首次进入顶点 u 的时间的 $d(u)$ 和结束对 u 的访问的时间 $f(u)$ 扩展到集合上, 有

$$d[U] = \min_{u \in U} \{d[u]\} \quad f[U] = \max_{u \in U} \{f[u]\}$$
- 引理2 设 C 和 C' 是有向图 $G=(V,E)$ 的两个不同连通分枝且存在 $(u, v) \in E$, 其中 $u \in C$ 且 $v \in C'$. 则 $f(C) > f(C')$.

证明: 分两种情况讨论

(1) $d(C) < d(C')$. 设 $x \in C$ 为第一个被发现的顶点, 则 $d[x] = d(C)$. $d[x]$ 时, C 中其他顶点和 C' 中所有顶点都是白色的。对任意的 $w \in C'$, 由 $x \rightarrow u \rightarrow v \rightarrow w$, 知 $f(x) > f(w)$. 所以, $f(C) > f(C')$.

(2) $d(C) > d(C')$. 设 $y \in C'$ 为第一个被发现的顶点, 则 $d[y] = d(C')$. $d[y]$ 时, C' 中其他顶点都是白色的, 因此 $f(C') = f[y]$. 由于 C' 中任一点都不可达 C (否则的话 $C \cup C'$ 强连通)。故 $f(C) > f(C')$.

HIT CS&E

推论3 设 C 和 C' 是有向图 $G=(V, E)$ 中的两个不同的连通分枝。如

果 $(u, v) \in E^T$, 其中 $u \in C$ 且 $v \in C'$, 则 $f(C) < f(C')$ 。

证明: $(u, v) \in E^T \Leftrightarrow (v, u) \in E$; 由引理2即可得到结论。
推论3意味着什么?

如果 u 的 $f(u)$ 最大且 $u \in C$ 则从 u 出发DFS不能访问到其他分枝
除非结束对 C 的访问后再重新指定DFS的出发点

定理 算法 `STRONGLY_CONNECTED_COMPONENTS(G)` 能够正确计算有向图 $G=(V, E)$ 所有连通分枝。

证明: 根据推论3对连通分枝数量做数学归纳法。自己下来书写!

时间复杂度 $O(|V|+|E|)$

两遍DFS, 外加一个计算 G^T 的时间开销

HIT CS&E

7.5 最小生成树算法

参见第五章, 贪心算法

HIT CS&E

7.6 单源最短路径算法

- 问题的定义
- 单源最短路径的子结构性质
- Bellman-Ford算法
- Dijkstra算法

HIT CS&E

单源最短路径问题

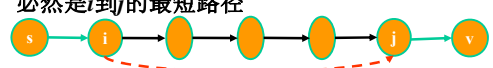
- 给定加权图或不加权的图 G , 找出给定的源顶点 s 到目标顶点 v 的最短路径
 - 在给定的网络拓扑下, 最小化路由代价。
 - 最小化基因-基因反应中的能量开销。
 - 最小化代价是许多实际问题中的基本要素
- 在加权图中
 - “最短路径” = 权值最小的路径
 - 路径的权值等于路径上所有边的权值之和
 - 不能用BFS来求解该问题
- 在不加权图中
 - “最短路径” = 边数最少的路径
 - 可以用BFS在 $O(V+E)$ 的时间内找出源顶点到目标顶点的最短路径。

HIT CS&E

最短路径的特征-优化子结构

优化子结构: 最短路径包含了最短子路径

设 s 到 v 的最短路径 P 经过顶点 i 和 j , 则 P 上从 i 到 j 的部分必然是 i 到 j 的最短路径



证明: 如果子路径 (i, j) 不是顶点 i 到顶点 j 的最短路径, 则

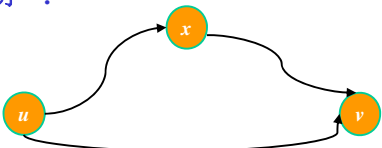
- 在 i 和 j 之间必然存在一条更短的路径(红色路径)
- 用这条更短的子路径替换原来的子路径
- 得到一条比原路径更短的路径。
- 矛盾。

HIT CS&E

最短路径的特征-三角不等式

- 定义 $\delta(u, v)$ 为从 u 到 v 的最短路径的代价(长度)
- 最短路径代价满足三角不等式

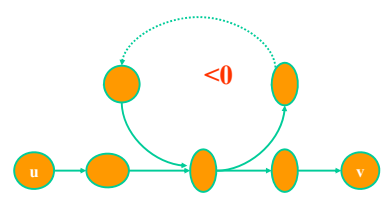
$$\delta(u, v) \leq \delta(u, x) + \delta(x, v)$$
- “证明”:



最短路径不比任何其他路径长

最短路径的特征-环

- u 到 v 的最短路径上能存在环吗?

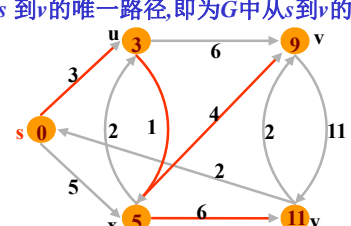


- 负环
如果 u 到 v 的最短路径上能存在负环, 则定义 $\delta(u,v)=-\infty$

最短路径的特征-最短路径树

最短路径树包含图 G 中从 s 出发可达的所有顶点

- 形成一棵以 s 为根的树.
- 在这棵树中从 s 到 v 的唯一路径,即为 G 中从 s 到 v 的最短路径.



最短路径的存储
 $\pi[v]$ 存储从 s 到 v 的最短路径中顶点 v 的前驱

最短路径的特征-松弛技术

松弛技术是最短路径算法的关键技术


- 对所有 v , 维护 $\delta(s,v)$ 的一个上界 $d[v]$

算法Relax(u,v,w)

Input 顶点 u 和 v , 图的加权函数 w

Output 松弛后的 $d[v]$

- if ($d[v] > d[u] + w(u,v)$) then
- $d[v] = d[u] + w(u,v)$;
- $\pi[v] = u$;



Bellman-Ford算法

算法 Bellman-Ford(G,w,s)

Input 图 $G=(V,E)$,边加权函数 w ,源顶点 s

Output s 到其所有可达顶点的最短路径

- For $\forall v \in V$ do
- $d[v] \leftarrow \infty$;
- $\pi[v] \leftarrow \text{null}$;
- $d[s] \leftarrow 0$;
- For $i \leftarrow 1$ to $|V|-1$ do
- For $\forall uv \in E$ do
- Relax(u,v,w);
- For $\forall uv \in E$ do
- If $d(v) > d(u) + w(u,v)$ then
- return FALSE
- Return TRUE

初始化

求解

执行 $|V|-1$ 遍, 松弛每条边

检查解的合理性, 环, 负环

请你确定Bellman-Ford算法的时间复杂度? $O(VE)$

Bellman-Ford算法的分析

算法 Bellman-Ford(G,w,s)

Input 图 $G=(V,E)$,边加权函数 w ,源顶点 s

Output s 到其所有可达顶点的最短路径

- For $\forall v \in V$ do
- $d[v] \leftarrow \infty$;
- $\pi[v] \leftarrow \text{null}$;
- $d[s] \leftarrow 0$;
- For $i \leftarrow 1$ to $|V|-1$ do
- For $\forall uv \in E$ do
- Relax(u,v,w);
- For $\forall uv \in E$ do
- If $d(v) > d(u) + w(u,v)$ then
- return FALSE
- Return TRUE

为什么算法运行 $|V|-1$ 遍就足够了?

- $d[v] = \delta(s,v)$ 必然在 $|V|-1$ 遍运行后成立
 - 设 $P = (v_0, \dots, v_k)$ 是从 s 到 v 最短路径, 其中 $v_0 = s$ 且 $v_k = v$.
 - 由于 P 简单路径, 故 $k < |V|-1$.
- 下面通过对最短路径的长度做归纳, 证明 $d[v_k] = \delta(s,v_k)$ 在 k 遍之后成立, 其中 k 是 (v_0, v_1, \dots, v_k) , 即 s 到 v_k 的最短路径的长度
 - 注意到最短 k -路径可以通过最短 $(k-1)$ -路径来构造

HIT CS&E

证明：考虑从s到v的最短路径

$s=v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_{k-1} \rightarrow v_k=v$

- 最初, $d[v_0]=d[s]=0=\delta(s,s)$ 且以后不再变化.
- 一遍之后, $d[v_1]=\delta(s,v_1)$ 是s到 v_1 的最短路径, 且 $d[v_1]$ 以后不再变化.
- 设 $k-1$ 边后有 $d[v_{k-1}]=\delta(s,v_{k-1})$, 则第 k 遍过程中,
 - 如果 $(d[v_k] > d[v_{k-1}] + w)$ 则 $d[v_k]=d[v_{k-1}]+w$
 - $d[v_k]=\delta(s,v_k)$, 因为每条最短 $(k-1)$ -路径均会被松弛过程检查和扩展.
- $d[v]=\delta(s,v)$ 在 $|V|-1$ 遍后成立.

HIT CS&E

Bellman-Ford算法具有负环检测能力

算法 Bellman-Ford(G, w, s)

Input 图 $G=(V, E)$, 边加权函数 w , 源顶点 s

Output s到其所有可达顶点的最短路径

```

1. For  $\forall v \in V$  do
2.    $d[v] \leftarrow \infty$ ;
3.    $\pi[v] \leftarrow \text{null}$ ;
4.  $d[s] \leftarrow 0$ ;
5. For  $i \leftarrow 1$  to  $|V|-1$  do
6.   For  $\forall uv \in E$  do
7.     Relax( $u, v, w$ );
8. For  $\forall uv \in E$  do
9.   If  $d(v) > d(u) + w(u, v)$  then
10.    return FALSE
11. Return TRUE
  
```

为什么算法能够发现负环?

HIT CS&E

- 在 G 中, 如果从 s 出发不能到达任何负环, 则
 - Bellman-Ford算法返回 TRUE,
 - $d[v]=\delta(s,v)$ 对任意顶点 v 成立.
- 在 G 中如果从 s 出发能够到达某个负环, 则
 - 算法返回 FALSE.
 - Bellman-Ford 能够检测负环的存在性.

证明:(1) 设 G 中没有负环.

- $|V|-1$ 遍后, 我们有 $d[v]=\delta(s,v)$ 对任意顶点 v 成立.
- 由三角不等式,

$$d[v]=\delta(s,v) \leq \delta(s,u) + w(u,v) = d[u] + w(u,v), \text{ 对任意 } (u,v) \in E \text{ 成立.}$$

HIT CS&E

证明:(2) 反证法

- 设 G 中从 s 可以到达负环 (v_0, v_1, \dots, v_k) 其中 $v_k=v_0$.

$$\sum_{i=1}^k w(v_{i-1}, v_i) < 0$$
- 但, 算法返回 TRUE
 - 没有负环被检测到
 - 检测负环 (v_0, v_1, \dots, v_k) 上的任意一条边时, 均有

$$d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i) \quad \text{for } i = 1, 2, \dots, k.$$

$$\sum_{i=1}^k d[v_i] \leq \sum_{i=1}^k d[v_{i-1}] + \sum_{i=1}^k w(v_{i-1}, v_i)$$

Since $\sum_{i=1}^k d[v_i] = \sum_{i=1}^k d[v_{i-1}]$, $\sum_{i=1}^k w(v_{i-1}, v_i) \geq 0$

矛盾

HIT CS&E

DAG上的单源最短路径

- 问题:** 在无环有向图(Directed Acyclic Graph-DAG)中如何高效地解决单源最短路径问题
 - Bellman-Ford算法的时间开销为 $O(VE)$
 - 在 DAG 中我们能否更快?
- Bellman-Ford算法执行 $|V|-1$ 遍
 - 每遍均需扫描所有边一遍
 - 对许多边的扫描均是无用的
- 事实上
 - 无需扫描不影响结果的边
 - 对于已经找到的最短路径, 其上的边无需再扫描

HIT CS&E

DAG上的单源最短路径

- 问题:** 在无环有向图(Directed Acyclic Graph-DAG)中如何高效地解决单源最短路径问题
 - Bellman-Ford算法的时间开销为 $O(VE)$
 - 在 DAG 中我们能否更快?
- 基本想法—利用拓扑排序
 - DAG中每条路径均是拓扑序顶点序列的子序列
 - 能够容易识别从 s 可达的顶点, 避免无用边扫描
 - 按照拓扑序处理顶点, 将始终是前向地处理每条路径, 避免重复扫描已知最短路径上的边
 - 仅需要一遍扫描

HIT
CS&E

DAG单源最短路径算法

算法 DAG-Shortest-Paths(G, w, s)

Input 无环有向图 $G=(V, E)$, 边加权函数 w , 源顶点 s

Output s 到其所有可达顶点的最短路径

1. 将 V 中顶点进行拓扑排序
2. For $\forall v \in V$ do
3. $d[v] \leftarrow \infty$;
4. $\pi[v] \leftarrow \text{null}$;
5. $d[s] \leftarrow 0$;
6. For each $u \in V$ (按拓扑序考虑) do
7. For $\forall v \in \text{Adj}[u]$ do
8. Relax(u, v, w);

大家尝试自己去分析该算法

HIT
CS&E

Dijkstra算法

- Dijkstra算法假设 $w(uv) \geq 0$ 对 $\forall uv \in E$ 成立
- 始终维护顶点集 S 使得
 - $\forall v \in S, d[v] = \delta(s, v)$, 即, s 到 v 的最短路径已经找到.
 - 初始值: $S = \emptyset, d[s] = 0$ 且 $d[v] = +\infty$
- 算法运行过程中
 - (a) 选择 $u \in V - S$ 使得

$$d[u] = \min \{d[x] \mid x \in V - S\}$$
 令 $S = S \cup \{u\}$
 此时 $d[u] = \delta(s, u)$!为什么?
 - (b) 对于 u 的每个相邻顶点 v 执行 RELAX(u, v, w)
- 重复上述步骤(a)和(b) 直到 $S = V$.
- 该算法类似与Prim算法, 属于贪心算法

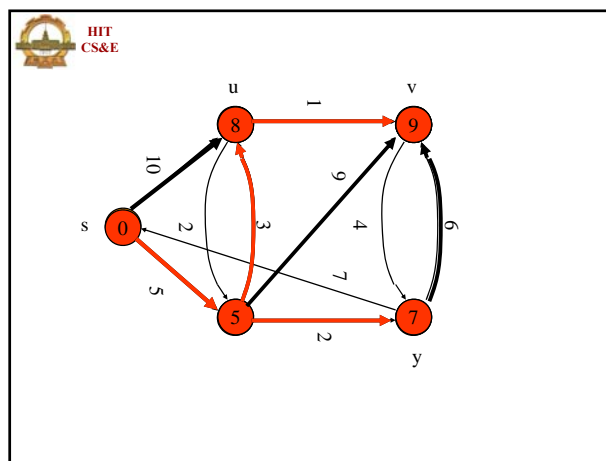
HIT
CS&E

算法 Dijkstra(G, w, s)

Input 图 $G=(V, E)$, 边加权函数 w , 源顶点 s

Output s 到其所有可达顶点的最短路径

1. For $\forall v \in V$ do
2. $d[v] \leftarrow \infty$;
3. $\pi[v] \leftarrow \text{null}$;
4. $d[s] \leftarrow 0$;
5. $S \leftarrow \emptyset$
6. $Q \leftarrow V$
7. while $Q \neq \emptyset$ do
8. $u \leftarrow \text{EXTRACT-MIN}(Q)$;
9. $S \leftarrow S \cup \{u\}$
10. For $\forall v \in \text{Adj}[u]$ do
11. Relax(u, v, w);



HIT
CS&E

第一步: 假设EXTRACT-MIN(Q)= x .

- sx 是仅含一条边的最短路径
 - 为什么?
 - 因为 sx 是从 s 出发的最短的边.
- 它也是 s 到 x 的最短路径

证明:

- (1) 设 $P: s \rightarrow u \dots \rightarrow x$ 是 s 到 x 的最短路径, 则 $w(s, u) \geq w(s, x)$.
- (2) 由于图中没有负权值边, 路径 P 的总权值至少为

$$w(s, u) \geq w(s, x).$$
- (3) 故, 边 sx 是 s 到 x 的最短路径.

HIT
CS&E

第二步: $S = \{s, x\}$ $d[y] = \min_{v \in V - S} d[v]$

- 论断: $d[y]$ 是从 s 到 y 的最短路径代价, 即
 - 要么 sy 是最短路径
 - 要么 $s \rightarrow x \rightarrow y$ 是最短路径.
- 为什么?
 - 如果 sy 是最短路径, 论断成立
 - 考察 $s \rightarrow x \rightarrow y$ 是从 s 到 y 的最短路径的情况

证明: (反证法) 设 $s \rightarrow x \rightarrow y$ 不是从 s 到 y 的最短路径

- (1) 设 $P_1: s \rightarrow y' \dots \rightarrow x$ 是 s 到 y 的最短路径, 其中 $y' \notin S$. (注意此时, 我们已经考察了 $y' = x$ 和 $y' = s$ 的情形).
- (2) 因此, $w(P_1) < w(s \rightarrow x \rightarrow y)$.
- (3) 由于 $w(uv) \geq 0$ 对任意边成立, 故 $w(sy') < w(P_1) < w(s \rightarrow x \rightarrow y)$. 进而 $d[y'] < d[y]$, 这样算法第二步不可能选中 y , 矛盾!

后续步骤: 设 S 是算法维护的集合, 令 $d[y] = \min_{v \in V-S} d[v]$

- 定理: $d[y]$ 是从 s 到 y 的最短路径代价 (正确性分析中最难的部分)

证明: (归纳法+反证)

归纳假设: 设对 $\forall v \in S$, $d[v]$ 是从 s 到 v 的最短路径的代价, 往证本次操作完成后 $d[y]$ 将是 s 到 y 的最短路径的代价

若不然, $d[y]$ 不是从 s 到 y 的最短路径的代价。设 P_1 :

$s \rightarrow \dots \rightarrow y' \rightarrow \dots \rightarrow y$ 是从 s 到 y 的最短路径, 其中 $y' \notin S$ 是 P_1 上第一个不属于 S 的顶点. 这意味着 $y \neq y'$ 且 $w(P_1) < d[y]$.

因此, $w(s \rightarrow \dots \rightarrow y') < w(P_1)$. (每条边的权值均非负)

进而 $w(s \rightarrow \dots \rightarrow y') < w(P_1) < d[y]$.

据此, $d[y'] \leq w(s \rightarrow \dots \rightarrow y') < w(P_1) < d[y]$.

因此, 算法在本次操作中不会选中 y , 矛盾!



Dijkstra算法的时间复杂度

- 时间复杂度依赖于优先队列 Q 的实现
- 模型1: 利用数组存储 Q
 - EXTRACT-MIN(Q) — 需要 $O(|V|)$ 时间.
 - 总共需要执行 $|V|$ 次 EXTRACT-MIN(Q).
 - $|V|$ 次 EXTRACT-MIN(Q)操作的总时间为 $O(|V|^2)$.
 - RELAX(u, v, w) — 需要 $O(1)$ 时间.
 - 总共需要执行 $|E|$ 次 RELAX(u, v, w)操作.
 - $|E|$ 次 RELAX(u, v, w)操作的总时间为 $O(|E|)$.
 - 总时间开销为 $O(|V|^2 + |E|) = O(|V|^2)$
- 模型2: Q 用堆实现.
 - 需要 $O(\log |V|)$ 时间.
 - 总时间开销为 $O(|V| \log |V| + |E|)$.



7.7 all-pairs shortest paths

- 问题的定义
- 单源最短路径的子结构性质
- Bellman-Ford算法
- Dijkstra算法



7.7.1 问题定义及求解方法

- 给定加权图或不加权的图 $G=(V, E)$, 我们对任意 $u, v \in V$ 计算出从 u 到 v 的最短路径
- 用Bellman-Ford算法或Dijkstra算法解决
 - 直接调用Bellman-Ford或Dijkstra算法 $|V|$ 遍
 - Dijkstra算法 $O(V \log V + E) \Rightarrow O(V^3)$
 - Bellman-Ford算法 $O(VE) \Rightarrow O(V^4)$
- Faster-All-Pairs-Shortest-Paths
 - $O(V^3 \lg V)$



最短路径的结构

7.7.2 基于矩阵乘法的算法

- 设 p 是顶点 i 到顶点 j 的最短路径
- 如果 $i=j$, 则 p 中不含任何边, 路径的权值为0
- 如果 $i \neq j$, 设 j 在路径 p 中的前驱为 k , 则 p 可以分解为 $i \rightarrow \dots \rightarrow k \rightarrow j$
 - 由最短路径的优化子结构知道 $i \rightarrow \dots \rightarrow k$ 是 i 到 k 的最短路径
 - 从而 $\delta(i, j) = \delta(i, k) + w(k, j)$
 - 如果 p 有 m 条边, 则 p' 有 $m-1$ 条边
 - 提示我们, 最短路径可以存为前驱矩阵 $\pi[i, j]$
 - $\pi[i, j]$ 表示从 i 到 j 的最短路径中 j 的前驱
 - 根据前驱矩阵, 可以打印所有的最短路径 (自己写个算法)
- 如果从路径的长度入手可能建立递归过程



递归计算

- 定义 $L_{ij}^{(m)}$ 是从 i 到 j 的至多仅含 m 条边的最短路径的代价
- 显然 $L_{ij}^{(0)} = 0$ if $i=j$ 或 $L_{ij}^{(0)} = \infty$ if $i \neq j$
- 因此 $L_{ij}^{(m)} = \min \{ L_{ij}^{(m-1)}, \min_{1 \leq k \leq n} \{ L_{ik}^{(m-1)} + w(k, j) \} \}$

$$= \min_{1 \leq k \leq n} \{ L_{ik}^{(m-1)} + w(k, j) \}$$
 因为 $w(j, j)=0$
- 由于从 i 到 j 的最短路径最多含有 $n-1$ 条边, 故

$$\delta(i, j) = L_{ij}^{(n-1)} = L_{ij}^{(n)} = L_{ij}^{(n+1)} = \dots$$
- 自底向上计算
 - $L^{(1)}, L^{(2)}, \dots, L^{(n-1)}$, 其中 $L^{(m)} = (L_{ij}^{(m)})_{n \times n}$
 - 注意 $L^{(1)} = W$ 是权值矩阵

HIT CS&E

算法 Extended_Shortest_path(L,w)
 Input 当前代价矩阵L,边加权函数w
 Output 经一步扩展后的代价矩阵

1. $n \leftarrow \text{row}[L];$
2. $L' \leftarrow (l'_{ij})_{n \times n}$
3. For $i \leftarrow 1$ to n do
4. For $j \leftarrow 1$ to n do
5. $l'_{ij} \leftarrow \infty$
6. For $k \leftarrow 1$ to n do
7. $l'_{ij} \leftarrow \min(l'_{ij}, l_{ik} + w(k,j))$
8. Return L'

矩阵乘法LW \rightarrow 替换 $\rightarrow \min$ $\rightarrow +$ $\rightarrow +$ Extended_Shortest_Path(L,W)

$L^{(1)} = L^{(0)} \cdot W = W$
 $L^{(2)} = L^{(1)} \cdot W = W^2$
 $L^{(3)} = L^{(2)} \cdot W = W^3$
 \vdots
 $L^{(n-1)} = L^{(n-2)} \cdot W = W^{n-1}$

时间复杂度 $O(n^4)$

修改算法完成对前驱矩阵的计算?

算法 Slow_All_Pairs_Shortest_path(W)
 Input 图的边加权函数矩阵w
 Output all_pair_Shortest_Paths代价

1. $n \leftarrow \text{row}[L];$
2. $L^{(1)} \leftarrow W$
3. For $m \leftarrow 2$ to $n-1$ do
4. $L^{(m)} \leftarrow \text{Extended_Shortest_path}(L^{(m-1)}, W)$
5. Return $L^{(n-1)}$

- 该算法能否检测到图中的负环?
- 如何降低该算法的时间复杂度?

例

$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$

$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$

$L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$

$L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$

© DB-L

HIT CS&E

降低算法的时间复杂度

Improving the running time:

$L^{(1)} = W$
 $L^{(2)} = W^2 = W \cdot W$
 $L^{(4)} = W^4 = W^2 \cdot W^2$
 \vdots
 $L^{(2^{\lceil \log(n-1) \rceil})} = W^{2^{\lceil \log(n-1) \rceil}}$

i.e., using repeating squaring!

Time complexity: $O(n^3 \log n)$.

HIT CS&E

算法 Faster_All_Pairs_Shortest_path(W)
 Input 图的边加权函数矩阵w
 Output all_pair_Shortest_Paths代价

1. $n \leftarrow \text{row}[L];$
2. $L^{(1)} \leftarrow W$
3. while $m < n-1$ do
4. $L^{(m)} \leftarrow \text{Extended_Shortest_path}(L^{(m-1)}, L^{(m-1)})$
5. $m \leftarrow 2m$
5. Return $L^{(m)}$

注意: $L^{(n-1)} = L^{(n)} = \dots$

HIT CS&E

7.7.3 Floyd-Warshall算法

- 动态规划算法求解all-pairs最短路径
 - 遵循动态规划算法设计的一般过程
 - 运行时间为 $O(V^3)$
 - 允许有负权值的边
 - 但不允许有负环
- 给出一个类似的算法寻找有向图的传递闭包

HIT CS&E

优化子结构

- 设 i 到 j 的最短路径为 $p: i \rightarrow \dots \rightarrow k \rightarrow \dots \rightarrow j$
 - 矩阵算法的优化子结构考虑 j 在 p 上的前驱
 - Floyd-Warshall算法的优化子结构将考虑路径 p 上经过的中间结点集
- 设 $V = \{1, 2, \dots, n\}$ $A_k = \{1, 2, \dots, k\} \subseteq V$
 - p 是从 i 到 j 的中间结点全属于 A_k 的最短路径
 - p' 是从 i 到 j 的中间结点全属于 A_{k-1} 的最短路径
 - Floyd-Warshall算法通过考察 p 和 p' 之间的关系建立优化子结构

HIT CS&E

$P: A_{k-1}$

如果路径 p 的中间结点全属于 A_{k-1} , 则其中间结点也全属于 A_k

如果路径 p 的中间结点全属于 A_k 且包含了结点 k

- p 可以从 k 处断开成两条路径
- $p_1: i \rightarrow \dots \rightarrow k$ 是从 i 到 k 的中间结点全属于 A_{k-1} 的最短路径
- $p_2: k \rightarrow \dots \rightarrow j$ 是从 k 到 j 的中间结点全属于 A_{k-1} 的最短路径

HIT CS&E

例

$i=4, j=5$

$k=0$

$k=1$

$k=2$

$k=3$

HIT CS&E

递归关系的建立

- 令 $d_{ij}^{(k)}$ = 从 i 到 j 的中间结点全属于 A_k 的最短路径的代价
 - $d_{ij}^{(k)} = w(i,j)$ if $k=0$ (此时路径至多一条边)
 - $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$ if $k \geq 1$
 - $D^{(n)} = (d_{ij}^{(n)})_{n \times n}$ 给出最终答案, $d_{ij}^{(n)} = \delta(i,j)$

HIT CS&E

Floyd-Warshall算法

算法Floyd_Warshall(W)

Input 图的边加权函数矩阵 W

Output all_pair_Shortest_Paths代价

1. $n \leftarrow \text{row}[L];$
2. $D^{(0)} \leftarrow W$
3. For $k \leftarrow 1$ to n do
4. For $i \leftarrow 1$ to n do
5. For $j \leftarrow 1$ to n do
6. $d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
5. Return $D^{(n)}$

定义前驱矩阵 $\pi^{(k)} = (\pi_{ij}^{(k)})_{n \times n}$

- 如何在计算 $D^{(k)}$ 的过程中, 完成 $\pi^{(k)}$ 的计算
- 如何由 $\pi^{(n)}$ 给出最短路径

HIT CS&E

有向图的传递闭包

- 给定有向图 $G=(V,E)$, 其中 $V = \{1, 2, \dots, n\}$
 - $\forall i, j \in V$, 在 G 中从 i 到 j 是否有有向路径可达
 - G 的传递闭包是有向图 $G^*=(V, E^*)$, 其中 $E^* = \{(i,j) | \text{在} G \text{中存在从} i \text{到} j \text{的有向路径}\}$
 - 令 G 中每条边的权值为1, 调用Floyd-Warshall算法可以计算传递闭包