H3 框架

01. 问题形式化

将问题化为Input,Output的形式

02. 设计算法

语言描述算法思想，伪代码

03. 分析算法

- 算法正确性
- 时间复杂度
- 空间复杂度

H3 渐进记法

渐进上界--大O表示法

渐进下界--大Ω表示法

大Θ表示法

掌握三种记法的概念即可

基本思想

分而治之将问题分解为`互不相交`的子问题，`递归`地求解子问题，再将它们的解组合起来，求出原问题的解.

问题实例

归并排序

问题形式化

Input：n个数的序列<a1,a2,...,an>

Output：输入序列的一个排列<a1',a2',...,an'>，满足$a_i \leq a_{i+1}(i \in Z^+)$

算法思想

将数组分为两个子数组，分别进行排序，最后合并

伪代码

```
mergeSort(A,p,r)
    if p < r
        q=floor((p+r)/2)
        mergeSort(A,p,q)
        mergeSort(q+1,r)
    Merge(A,p,q,r)
```

算法分析

时间复杂度

当n=1时

$$T(n) = \Theta(1)$$

当n>1时

$$T(n) = 2(T/n) + \Theta(n)$$

根据递归方程画出递归树，可以求得分解问题的时间复杂度

$$cnlgn$$

合并的时间复杂度为cn

则时间复杂度为

$$cn(1 + lgn)$$

$$O(nlogn)$$

空间复杂度

$$O(n)$$

合并时需要额外的规模为n的数组空间

H4 最大子数组问题

问题实例化

Input：一个含有n个数的数组A[1...n]

Output：(1)索引i,j使得子数组A[i..j]的和在子数组中最大(2)A[i...j]的和

算法设计

01. 暴力求解 时间复杂度O(n^2)

伪代码省略，基本思想是遍历每一个子数组，记录最大子数组的和和下标

02. 分而治之

通过分治的思想，我们很容易想到将问题分解为两个子问题和一个`特殊问题`

- 左数组的最大子数组
- 右数组的最大子数组
- 经过数组中点的最大子数组(特殊问题)

特殊问题的求解

暴力求解

算法思想：从数组的中点开始向左(右)遍历，记录往左(右)的最大数组和的下标

```
Find-Maximum-Crossing-Subarray(A,low,mid,high)
left-sum=-∞
sum=0
for i=mid to low
sum=sum+A[i]
if sum>left-sum
left-sum=sum
max-left=i


right-sum=-∞
sum=0
for j=mid+1 to high
sum=sum+A[j]
if sum > right-sum
    right-sum=sum
    max-right=j

   return (max-left,max-right,left-sum+right-sum)
```

时间复杂度 O(n)

```
Find-Maximum-Subarray(A,low,high)
 if low==high
     return(low,high,A[low])
 else mid=(low+high)/2
 (left-low,left-high,left-sum)=Find-Maximum-Subarray(A,low,mid)
 (right-low,right-high,right-sum)=Find-Maximum-Subarray(A,mid+1,high)
 (cross-low,cross-high,cross-sum)=Find-Maximum-Crossing-Subarray(A,low,mid,high)
 if(left-sum≥right-sum&&left-sum≥cross-sum) return (left-low,left-high,left-sum)
   ... 以此类推，返回最大值对应的子数组下标和子数组和
```

通过递归树求解递归方程可以得到

算法时间复杂度为 O(nlogn)

将函数返回值修改为(max-subarray-sum,max-prefix-sum,max-suffix-sum,total sum)

通过左数组的最大后缀和右数组的最大前缀就可以计算出经过中点的最大子数组和

伪代码

```
Find-Maximum-Subarray-Fast(A,l,r)
if l>r then return (0,0,0,0)
else if l==r then return (A[r],A[r],A[r],A[r])
else
    mid=(l+r)/2
    (m1,p1,s1,t1)=Find-Maximum-Subarray-Fast(A,l.mid)
    (m2,p2,s2,t2)=Find-Maximum-Subarray-Fast(A,mid+1,r)
max-subarray-sum=max(s1+p1,m1,m2)
max-prefix=max(p1,t1+p2)
max-suffix=max(s2,s1+t2)
total-sum=t1+t2
return (max-subarray-sum,max-prefix,max-suffix,total-sum)
```

时间复杂度 O(n)

基本思想

动态规划应用于子问题重叠的情况，避免子问题的重复计算

一个动态规划算法的设计通常有下面四个步骤：

- 刻画一个最优解特征
- 递归地定义最优解的值
- 计算最优解的值
- 利用计算出的信息构造一个最优解

问题实例

钢条切割问题

问题形式化

Input：钢条的总长度n，长度为i的钢条的价值pi

Output：将钢条切割后的总价值 rn

算法设计

一般地，对于总价值r_n，我们可以用更短的钢条的最优切割收益来描述

$$r_n = max(p_n, r_1 + r_{n-1}, \ldots, r_{n-1} + r_1)$$

我们称钢条切割问题满足`最优子结构`性质，即问题的最优解由相关子问题的最优解组合而成，而这些子问题可以独立求解

`自顶向下递归求解`

我们可以将上面的公式简化为

$$r_n = max_{1\leq i\leq n}\{p_i + r_{n-1}\}$$

根据这个公式我们可以得出自顶向下递归求解该问题的伪代码

```
Cut-Rod(p,n)
    if n==0
        return 0
    q=-∞
    for i=1 to n
        q=max(q,p[i]+Cut-Rod(p,n-i))
return q
```

时间复杂度分析

令T(n)表示第二个参数为n时，Cut-Rod的调用次数

$$T(n) = 1 + \sum_{j=0}^{n-1} Tj$$

求解这个递归方程

$$T(n) = 2^n$$

算法时间复杂度为O(2^n)

为什么递归方法时间复杂度这么高？ 重复计算了相同的子问题

动态规划如何解决？ 仔细安排求解顺序，对每个子问题只求解一次，并将结果保存下来，以便下次使用

两种动态规划实现方法

带备忘的自顶向下法

```
Memoized-Cut-Rod(p,n)
    r[0...n]为新数组，用于存放已经解决的子问题的解
    for i=0 to n
    r[i]=-∞
```

```
return Memoized-Cut-Rod-Aux(p,n,r)

Memoized-Cut-Rod-Aux(p,n,r)
if r[n]≥0 return r[n]
if n==0
    q==0
else q=-∞
    for i=1 to n
        q=max(q,p[i]+Memoized-Cut-Rod(p,n-i,r))
r[n]=q
return q
```

自底向上法

```
Memoized-Cut-Rod(p,n)
    r[0, ... ,n]为一个新数组
    for i=0 to n
    r[0]=0
    for j=1 to n
        q=-∞
        for i=1 to j
            q=max(q,p[i]+r[j-i])
        r[j]=q
return r[n]
```

两种方法的时间复杂度均为$\Theta(n^2)$

子问题图

表示问题与其子问题之间的依赖关系

矩阵链乘法

问题形式化

Input：一个矩阵链(A1,A2,...,An) Ai的维度为Pi-1xPi

Output：一个完全括号化方案，使得计算乘积A1A2...An所需的标量乘法数量最少

算法设计

暴力求解，枚举每一种可能

$$P(n)=\begin{cases}1, & \text{if} \quad n=1\\ \sum_{k=1}^{n-1}P(k)P(n-k), & \text{if} \quad n\geq 2\end{cases}$$

暴力求解的时间复杂度为$\Omega(2^n)$

应用动态规划方法

定义

m[i,j]是计算$A_{i\ldots j}$，矩阵链Ai到Aj的最小标量乘法次数

$$m[i,j]=\begin{cases}0 & \text{if} \quad i=j\\ min_{i\leq k\leq j}(m[i,k]+m[k+1,j]+p_{i-1}p_kp_j) & \text{if} \quad i<j\end{cases}$$

带备忘的自顶向下求解

```
Memoized-Matrix-Chain(P)
    n=P.length-1
m[1..n,1..n]为一个新表
for i=1 to n
    for j=1 to n
        m[i,j]=∞
return Lookup-Chain(m,p,1,n)

LookUp-Chain(m,p,i,j)
        if m[i,j]<∞
            return m[i,j]
if i==j m[i][j]=0
else for k=i to j-1
    q=Lookup-Chain(m,p,i,k)+Lookup-Chain(m,p,k+1,j)+pi-1pkpj
    if q< m[i,j]
        m[i.j]=q
return m[i,j]
```

自底向上求解

```
Matrix-Chain-Order(p)
    n=p.length-1
```

```
    let m[1 ... n,1 ... n],s[1 ... n,1 ... n]为新表
for i=1 to n
    m[i,i]=0
for l=2 to n
    for i=1 to n-l+1
        j=i+l-1
        m[i][j]=∞
        for k=i to j-1
            q=m[i,k]+m[k+1,j]+pi-1pkpj
            if q<m[i,j]
                m[i,j]=q
                s[i,j]=k
return m and s
```

H4 最长公共子序列问题

问题形式化

Input：两个序列X(x1,x2,...,xn),Y(y1,y2,...yn)

Output：最长公共子序列长度

定理 LCS的最优子结构

令X(x1,x2,...,xm),Y(y1,y2,...yn)为两个序列，Z(z1,z2,..,zk)为X，Y的任意LCS

01. 如果$x_m = y_n$,则$z_k = x_m = y_n$且Zk-1是Xm-1和Yn-1的一个LCS
02. 如果$x_m \neq y_n$,那么$z_k \neq x_m$意味着Z是Xm-1和Y的一个LCS
03. 如果$x_m \neq y_n$,那么$z_k \neq y_n$意味着Z是X和Yn-1的一个LCS

Bellman方程

c[i,j]表示Xi和Yj的LCS长度

$$c[i,j] = \begin{cases} 0 & \text{if} \quad i=0或j=0 \\ c[i-1,j-1]+1 & \text{if} \quad i,j>0且xi=yj \\ max(c[i,j-1],c[i-1,j]) & \text{if} \quad i,j>0且xi \neq yj \end{cases}$$

```
LCS-length(X,Y)
    m=x.length
    n=y.length
b[1 ... m,1 ... n],c[0 ... m,0 ... n]为新表
for  i to m
    c[i,0]=0
for j=0 to n
    c[0,j]=0
for  i=1 to m
    for j=1 to n
        if xi==yj
            c[i,j]=c[i-1,j-1]+1
            b[i,j]="↖"
        else if c[i-1,j]≥c[i,j-1]
            c[i,j]=c[i-1,j]
            b[i,j]="↑"
        else c[i,j]=c[i,j-1]
            b[i,j]="←"
return c and b
```

时间复杂度$O(nm)$

H3 动态规划原理

H4 最优子结构

使用动态规划求解最优化问题的第一步就是刻画最优解的结构，如果一个问题的最优解包含其子问题的最优解，我们就称此问题具有最优子结构性质。

常用反证法证明

H4 重叠子问题

问题的递归算法会反复地求解相同地子问题，而不是生成新的子问题。

使用递归算法反复求解相同的子问题，我们就称最优化问题具有重叠子问题性质

重构最优解

从实际考虑，我们通常将每个子问题所做的一个选择存在一个表里，这样就不必根据代价值来重构这些信息

备忘

我们在保持自顶向下的策略时，想要达到与自底向上动态规划方法相似的效率，向递归算法中加入备忘机制。维护一个表记录子问题的解，但仍保持递归算法的控制流程

基本思想

贪心算法在每一步都做出当时看起来最佳的选择，也就是说，做出局部最优的选择，寄希望于这样的选择能导致全局最优解

贪心算法原理

我们可以按照如下步骤设计贪心算法：

01. 将最优化问题转换为这样的形式：对其做出一次选择后，只剩下一个子问题需要求解
02. 证明做出贪心选择后，原问题总是存在最优解
03. 证明做出贪心选择后，剩余的子问题满足性质：其最优解与贪心选择组合即可得到原问题的最优解

贪心选择性质

当进行选择时，贪心算法直接做出在当前问题中看来最优的选择，而不必考虑子问题的解

最优子结构

最优子结构性质是是否应用贪心算法的关键要素

H3 问题实例

H4 活动选择问题

问题形式化

Input：集合S={a1,a2,...,an},每个活动ai对应的开始时间si,结束时间fi

Output：兼容活动的最大子集$S' \subseteq S$

最优子结构

我们容易证明活动选择问题具有最优子结构性质

贪心选择

直觉上，我们应该选择一个活动，选出它后剩下的资源能被尽量多的其他任务所利用。即选择结束时间最早的活动

定理

考虑任何非空子问题S_k,令a_m是S_k中结束时间最早的活动，则am在Sk的某个最大兼容活动子集中

贪心算法

```
Greedy-Activity-selector(s,f)
n=s.length
sort activity by finish times and renumber so that f1≤f2≤ ... ≤fn
A={a1}
k=1
for m=2 to n
    if s[m]≥f[k]
    A=A∪{am}
    k=m
return A
```

H3 Dijkstra's algorithm

问题形式化

Input：一个有向图G=(V,E),边长$w_e \geq 0$,源点s∈V

Output：从s到图中所有顶点的最短路

定义

a[u]：从源点s到u的最短路径长度

S：被访问的顶点的集合

$\pi[v]$：顶点v的前驱节点

贪心选择：从没有访问到的节点中选出a[u]值最小的顶点u

- 加入到S
- 对所有从u离开的边e=(u,v),如果a[v]>a[u]+w_e，则更新
    - a[v]<-a[u]+we
    - $\pi[v]$<-u

```
Dijkstra(V,E,w,s)
Foreach v≠s: a[v]←∞,Π[v]←null,a[s]=0
Create an empty priority queue PQ
Foreach v∈V:Insert(PQ,v,a[v])
while(is-not-empty(PQ))
    u←Extract-Min(PQ)
    Foreach edge e=(u,v) leaving u:
    If(a[v]>a[u]+we)
        Decrease-Key(PQ,v,a[v]+we)
        a[v]←a[u]+we
        Π[v]←u
```

堆的基本操作

sift-up

sift-down

insert(x)

Decrease/Increase-key(i,key)

Extract-Min/Max()

时间复杂度都为logn

算法正确性证明

数学归纳法

H3 背包问题

0-1背包问题使用贪心算法无法保证得出最优解

但是分数背包可以

0-1背包问题可以使用动态规划求解，但是是伪多项式时间内解决，是NP-hard问题

```
Knapsack(n,W,w1, ... ,wn,v1, ... ,vn)
for w=0 to W
    M[0,w]←0
for i=1 to n
    for w=0 to W
    if(wi>w) M[i,w]=M[i-1,w]
    else M[i,w]=max{M[i-1,w],M[i-1,w-wi]+vi}
return M[n,W]
```

H3 Bellman-Ford算法

问题实例化

Input：图G=(V,E)与边长l_{uv}，一个源点s

Output：s->v路径的最短路 $v \in V$

引理1：如果s->v的路径上存在负圈，那么不存在从s到v的最短路

引理2：如果G没有负圈，则存在从s到v的简单路径

问题分解

- 无负环路的单源最短路问题
- 负环路的存在性问题

无负环路的单源最短路问题

定义：

OPT(i，v)：从s到v的最多i个条边的最短路径长度

目标：OPT(i-1,v)

Bellman方程

$$OPT(i,v)=\begin{cases}0 & if \quad i=0\ and\ v=s\\ \infty & if \quad i=0\ and\ v\neq s\\ OPT(i-1,v) & if \quad \neg\exists(u,v)\in E\ or\ v=s\\ min(OPT(i-1,v),OPT(i-1,u)+l_{uv}) & if \quad i>0\exists(u,v)\in E\ and\ v\neq s\end{cases}$$

```
Shorteset-Path(V,E,l,s)
Foreach node v ∈V:
    M[0,v]←∞
M[0,s]←0
For i=1 To n-1
 Foreach node v∈V:
    M[i,v]←M[i-1,v]
    Foreach edge(u,v)∈E:
    M[i,v]←min{M[i,v],M[i-1,u]+luv}
```

算法复杂度分析

空间复杂度：$\Theta(n^2)$

时间复杂度：算法遍历所有顶点以及与之相关联的边，时间复杂度为O(mn)

如何构造最短路

01. 维护前序节点数组predecessor[i,v]

02. 对于最优长度M[i,v],只考虑M[i,v]=M[i,u]+luv这样的边，子图中任意有向路径都是最短路

空间复杂度优化

使用一维数组d[v]代替二维数组

d[v]=目前为止，从s到v路径的最短长度

```
Shorteset-Path(V,E,l,s)
Foreach node v ∈V:
    d[v]←∞
d[s]←0
For i=1 To n-1
 Foreach node v∈V:
    Foreach edge(u,v)∈E:
    d[v]←min{d[v],d[u]+luv}
```

引理3：对于每个顶点v，d[v]是单调不增的

引理4：在第i轮后，d[v]小于等于所有最多使用i个顶点的s到v路径

证明：

使用数学归纳法证明

归纳基础：i=0 时，明显满足

归纳假设：i=k时，假设所有最多经过k条边的s到v路径均为最短路径

当i=k+1，取P为任意s到v的最多含有k+1条边的路径

(u,v)为P中最后一条边，P'为P的s到u的子路

因为P'是最多含有k条边的路径，根据归纳假设

$$d[u] \leq l(P')$$

进而有

$$d[v] \leq d[u] + l_{uv}$$
$$d[v] \leq l(P') + l_{uv} = l(p)$$

如果最短路上有k个顶点，该算法会在k轮后找到它

定理2： 在没有负圈的情况下，Bellman-Ford算法计算出所有s到v的最短路所需的时间复杂度是O(mn)

空间复杂度是$\Theta(n)$

负圈的检测问题

引理5：

如果对于所有顶点v有

$$OPT(n,v) = OPT(n-1,v)$$

那么s到v的路径上没有负圈

引理6：

如果对所有的顶点有

$$OPT(n,v) < OPT(n-1,v)$$

那么从s到v的包含最多n个顶点的最短路径包含一个圈W，W是一个负圈

证明：

根据上式我们可以得出，存在s到v的路径P，P含有n个顶点，否则OPT(n,v)=OPT(n-1,v)

根据鸽巢原理，路径P上至少存在一个重复顶点x，换句话说，路径P上含有环路

如果该环路不是负圈，则删除W会得到一条最多有n-1个顶点的并且权值更小的路径，即

$$OPT(n,v) \geq OPT(n-1,v)$$

这与假设矛盾

所以W是负圈

定理4

OPT(n, v) = OPT(n − 1, v) for every node v, if and only if no negative cycles.

```
Bellman-Ford(V,E,l,s)
Foreach node v ∈V:
    d[v]←∞
```

```
    predecessor[v]←null
d[s]←0
For i=1 To n-1
 Foreach node v∈V:
    Foreach edge(u,v)∈E:
    d[v]←min{d[v],d[u]+luv}
    predecessor[v]←u
Foreach edge(u,v)∈E
    If(d[v]>d[u]+luv)
    Return True
Return False
```

定理5

我们可以在$\Theta(mn)$时间内，找到所有顶点的单源最短路径，或者找到一个负圈

H3 多源最短路径：按边数分解的（基于矩阵乘法的）动态规划算法。

不考虑负圈的情况

问题形式化

Input：一个nxn矩阵W代表一个含有n个顶点的有向图G=(V,E)的边的权值

$$W=(w_{ij})$$

$$w_{ij}=\begin{cases}0 & i=j\\ (i,j)\text{的权值} & i\neq j,(i,j)\in E\\ \infty & i\neq j,(i,j)\notin E\end{cases}$$

Output:一个nxn矩阵D=(d_{ij})，d_{ij}表示从i到j的最短路径长度

时间复杂度 O(n^4) 优化后 O(n^3logn)

基础类矩阵乘法算法

定义：

$$l_{ij}^{(m)} = 从i到j最多使用m条边的最短路径长度$$

目标：每一对顶点的$l_{ij}^{(n-1)}$

Bellman方程

$$l_{ij}^{(m)} = \begin{cases} 0 & m = 0 \\ \infty & m \neq 0, i \neq j \\ min_{1 \leq k \leq n}\{l_{ij}^{(m-1)} + w_{ij}\} & m > 0 \end{cases}$$

定义：

$L^{(m)}$=$(l_{ij}^{(m)})$ m=1,2.....,n-1

```
All-Pairs-Shortest-Paths(W)
n = W.rows
L(1)= W
for m=2 to n-1
    L(m)为一个新的nxn矩阵
    L(m)=Extend-Shortest-Paths(L(m-1),W)
return L(n-1)

Extend-Shortest-Path(L,W)
n=L.rows
让 L'=(l'ij)为一个新的nxn数组
for i=1 to n
    for j=1 to n
    l'ij=∞
    for k=1 to n
        l'ij=min(l'ij,lik+wkj)
return L'
```

时间复杂度O(n^4)

时间复杂度优化

```
All-Pairs-Shortest-Paths(W)
n = W.rows
L(1)= W
while m<n-1
    L(m)为一个新的nxn矩阵
    L(2m)=Extend-Shortest-Paths(L(m),L(m))
    m=2m
return L(m)
```

H3 多源最短路径：按顶点编号分解的（Floyd-Warshall）动态规划算法。

定义

$$d_{ij}^k = 从i到j经过中间顶点\{1,2,\ldots,k\}的最短路径长度$$

目标

$$每对顶点的d_{ij}^n$$

Bellman方程

$$d_{ij}^k = \begin{cases} w_{ij} & k=0 \\ mi\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & k>0 \end{cases}$$

```
Floyd-Warshall(W)
n=W.rows
D(0)=W
for k=1 to n
    让D(k)=(dij(k))为一个新的nxn矩阵
    for i=1 to n
        for j=1 to n
            dij(k)=min(dij(k-1),dik(k-1)+dkj(k-1))
return D(k)
```

H3 基本概念

流网络

流网络是一个有序对 G=(V,E,s,t,c)

- 有向图G=(V,E)和源点s和汇节点
- E中所有边的容量c(e)≥0

流f(e)

流f是一个E集合上的一个函数：

- 对于E中的每条边e (容量限制)

$$0 \leq f(e) \leq c(e)$$

- 对于除了源点和汇节点的点(流量守恒)

$$\sum_{e\ into\ v} f(e) = \sum_{e\ out\ of\ v} f(e)$$

流的值

$$val(f) = f^{out}(s) - f^{in}(s)$$

割

割是V集合的一个划分(A,B),其中$s \in A$且$t \in B$

割的容量

从A到B的边的容量之和

$$Cap(A,B) = \sum_{e\ out\ of\ A} c(e)$$

H3 最大流问题

问题形式化

Input：一个四元组G=(V,E,s,t,c)

Output：流的最大值max(val(f))

算法设计

贪心算法

- 初始化所有E中的边的流f(e)=0
- 找到一个s到t的路径P，P上每条边都满足

$$f(e) \leq c(e)$$

- 沿着路径P增大流量
- 重复以上步骤直到找不到可行的路径

保证能找到最优解吗？ 不能

why? 一旦增加了某条边的容量，就永远无法减少它

如何解决? 想办法撤销坏的决定?

一些概念

剩余网络 Residual network

$$G_f = (V, E_f, s, t, c_f)$$

剩余网络中将E中的原始边分为两种边：

原始边

$$e = (u, v) \in E$$
$$Flow\ f(e)$$
$$Capacity\ c(e)$$

前向边

$$e = (u, v)$$
$$e \in E \; and \; e \in E_f$$
$$c_f(e) = c(e) - f(e)$$

后向边

$$e^{reverse} = (v, u)$$
$$e^{reverse} \neq\in E \; and \; e^{reverse} \in E_f$$
$$c_f(e^{reverse}) = f(e)$$

增广路径

增广路径是剩余网络G_f中的一条简单路径

瓶颈容量

瓶颈容量是增广路径上的边的最小容量

Augment操作

将增广路径上的所有边的容量加上瓶颈容量

```
Augment(f,c,P)
δ←增广路径P的瓶颈容量
Foreach edge e ∈ P
 If(e∈E) f(e)← f(e)+δ
 ElseIf (e不属于E且e∈Ef) f(e)←f(e)-δ
 Return f
```

Ford-Fulkerson 算法

```
Foreach edge e ∈E: f(e)← 0
Gf← residual network of G with respect to flow f
While(Gf中存在一条从s到t的路径P)
    f←Augment(f,c,P)
    Update Gf
return f
```

H3 最大流与最小割的关系

流值引理

f为任一流且(A,B)为任一割，则流f的值等于经过割(A,B)的净流量

$$val(f) = \sum_{e\ out\ of\ A} f(e) - \sum_{e\ into\ A} f(e)$$

证明：

$$\begin{aligned} val(f) &= \sum_{e\ out\ of\ s} f(e) - \sum_{e\ into\ s} f(e) \\ &= \sum_{v \in V}(\sum_{e\ out\ of\ v} f(e) - \sum_{e\ into\ v} f(e)) \\ &= \sum_{e\ out\ of\ A} f(e) - \sum_{e\ into\ A} f(e) \end{aligned}$$

除了源点s外，其他顶点的流出量等于流入量

弱对偶

让f为任一流，(A,B)为任一割，则

$$val(f) \leq cut(A,B)$$

证明：

$$
\begin{aligned}
val(f) &= \sum_{e\ out\ of\ A} f(e) - \sum_{e\ into\ A} f(e)(\text{流值引理}) \\
&\leq \sum_{e\ out\ of\ A} f(e) \\
&\leq \sum_{e\ out\ of\ A} c(e) \\
&= cut(A,B)
\end{aligned}
$$

推论：让f为任一流，(A,B)为任一割，如果val(f)=cap(A,B)，则f是最大流且(A,B)是最小割

证明：

01. f是最大流

对于任一的流f'，根据弱对偶有

$$
val(f') \leq cut(A,B) = val(f)
$$

即f是最大流

02. cut(A,B)是最小割

对于任一的割cut(A',B')

$$
cut(A',B') \geq val(f) = cut(A,B)
$$

即cut(A,B)是最小割

最大流最小割定理

最大流的值等于最小割的容量

增广路径定理

流f是最大流，当且仅当没有增广路径

以下三条关于流f的定理等价：

01. 存在割cut(A,B)=val(f)
02. 是最大流

03. 没有关于f的增广路径

证明：

1->2 弱对偶

2->3 反证法

3->1

让f为一个没有增广路径的流

A为s所能到达的在剩余网络Gf中的所有顶点的集合

显然$s \in A, t \notin A$

对于边e=(u,v),$u \in B, v \in A$，一定有f(e)=0，否则u是s可以到达的顶点，这与假设矛盾

对于任意的边e=(u,v),$u \in A, v \in B$ 一定有f(e)=c(e),否则v是s可以到达的顶点，这与假设矛盾

进而有

$$
\begin{aligned}
val(f) &= \sum_{e\ out\ of\ A} f(e) - \sum_{e\ into\ A} f(e) \\
&= \sum_{e\ out\ of\ A} c(e) - 0 \\
&= cut(A, B)
\end{aligned}
$$

H3 算法分析

每条边的容量可以是整数或者有理数

如果在选择增广路径时作出了"坏"的选择，算法的表现可能会很差

当边的容量是无理数时，不能保证算法会终止，即算法可能不能收敛到最大流的值

假设边的容量为1到C之间的整数，则：

- 每条边的流f(e)h和剩余容量cf(e)都是整数(完整性不变量)

- Ford-Fulkson算法在最多val(f^*)次增广路径后结束，f^*是最大流(增广路径定理)
- 一定存在最大流f^* (完整性定理)

推论：Ford-Fulkerson算法的运行时间是O(mnC)

通过BFS或DFS可以在O(m)时间内找到一个增广路径

H3 最大流算法的应用：匹配

定义：

给出一个无向图G=(V,E),M为E的子集，如果每个顶点最多出现在M中的一条边上，则称M为一个匹配

最大匹配：给定一个图G，找到最大势的匹配

如果一个图G中的顶点可以分为两个子集L,R并且每条边都关联一个L中的顶点和一个R中的顶点，则称G为二分图

二分匹配：给出一个二分图，找到最大匹配

初始化：

- 创建一个有向图G'=(L∪R∪{s,t},E')
- 给从L到R的所有边分配无限的容量
- 给s到L中每个顶点的边分配单位容量
- 给R中每个顶点到t的边分配单位容量

定理：G的最大匹配的值=G’中最大流的值

反证法证明

H3 问题及其困难性(hardness)

问题及其实例

问题：一个问题由输入和输出部分组成

实例：一个具体的输入

判定形式和优化形式

问题的判别形式只需回答yes或no，优化形式输出问题的最优结果

判别形式的输出比较简单，利于我们进行复杂度分析

问题的两种形式之间的关系

只要问题的一种形式可解，另一种形式也可解

判别形式->优化形式 二分

优化形式->判别形式 施加结果值的界限

H3 归约 Reduction

找到两个关系之间的联系

一个可以在满足下列条件下能够将任意问题A的实例转化为一个问题B的实例的程序

条件：

- 转化：转化过程使用多项式时间完成
- 等价：两个实例的答案相同

记作$A \leq_P B$ 读作A可规约为B

定理

01. 如果B可以在多项式时间内解决，那么A在多项式时间内也可以解决
02. A是困难的，B也是困难的

1 根据归约的概念很容易证明 2是1的逆否命题，也可以用反证法证明

H3 3SAT问题到Clique问题的归约

3-CNF-SAT问题

Input:一个合取范式，包含k个子句，每个子句包含三个文字

$$\Phi = C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

Output：是否存在使该合取范式满足的真值指派

Clique问题

输入：一个无向图G=<V,E>和一个整数k

输出：是否存在一个顶点集V的含有k个顶点的子集V'使得V'中的任意两个顶点都由一条边所连接

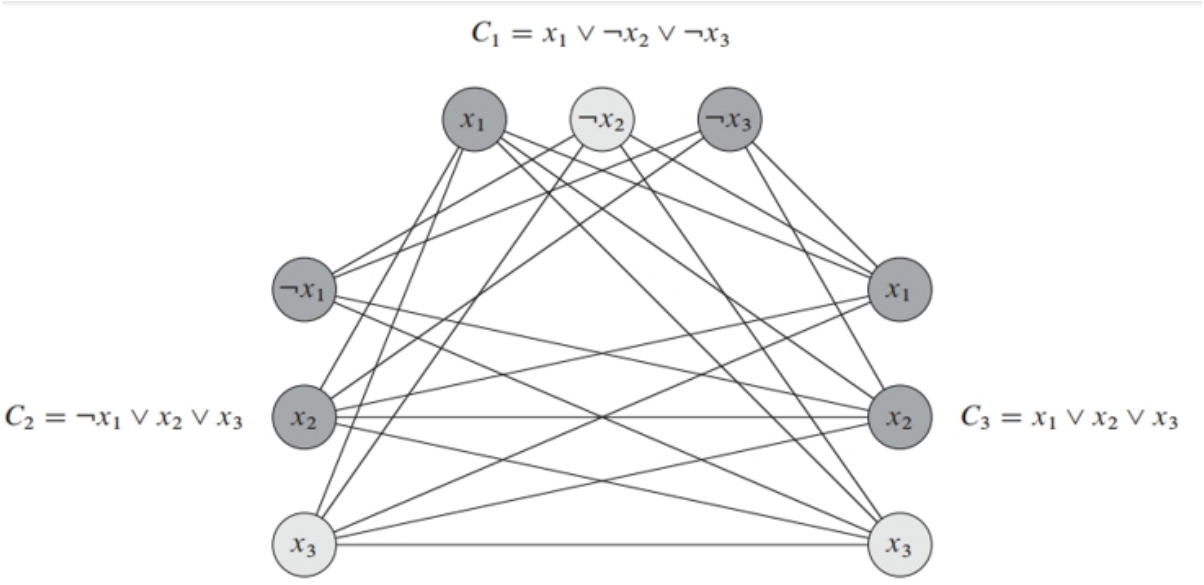
定理

$3SAT \leq_P Clique$

证明：

给出一个3SAT实例Φ,我们可以构造一个Clique实例<G,k>,且存在一个大小为k的团，当且仅当Φ可满足

- 为每个子句创建一个顶点子集，一个顶点代表一个文字
- 如果两个顶点属于不同的子句，并且它们不是彼此的否定，那么连接对应的两个顶点



现在证明

$$\text{存在一个大小为}k\text{的团} \iff \Phi\text{是可满足的}$$

$\Leftarrow$ ：

假定Φ有一个可满足性赋值，则每个子句C_r中至少有一个文字l_i^r的赋值为1，这个问题对应图中的顶点为v_i^r,从每个子句中选出一个赋值为1的文字，就得到k个顶点组成的集合V'，我们断言V'是一个团。因为对于任意两个不同的顶点u,v∈V'，u,v对应的文字不是互补的，也就是说(u,v)∈E ,进而V'是一个大小为K的团

$\Rightarrow$ ：

假设存在一个大小为k的团V'，G中没有连接同一个三元组中顶点的边，因此V'中恰好包含每个三元组的一个顶点。我们将每个满足v_i^r∈V'的问题l_i^r赋值为1，由于图G中两个互补文字之间不存在连线。这样每个子句都是可满足的，因此Φ也是可满足的

H3 团问题到顶点覆盖问题的归约

顶点覆盖问题

Input：一个无向图G=(V,E)

Output：一个最小的顶点子集$V' \in V$,满足如果有$(u,v) \in E$，则$u \in V'$或者$v \in V'$

定理

$$Clique\text{问题} \leq_q \text{顶点覆盖问题}$$

图G具有一个规模为k的团，当且仅当$\bar{G}$有一个规模为|V|-k的顶点覆盖

证明：

⇒:

当图G有一个规模为k的团V'，则$\bar{G}$中存在一个规模为|V|-k的顶点覆盖$\bar{V} = V - V'$

因为$\bar{G}$中所有边都至少与$\bar{V}$中一个顶点相关联，否则该边(假设为e)的两个端点都属于V'

V'的所有顶点构成一个完全子图，e在G补图中不存在

⇐:

当$\bar{G}$中有一个规模为|V|-k的顶点覆盖$\bar{V}$时，G中有一个规模为k的团V'

V'=V-$\bar{V}$,|V'|=k, 不存在一条边e=(u,v),$u, v \in V'$

否则$\bar{V}$将不是一个顶点覆盖，根据补图的定义在G中 V'中顶点两两相邻，即就是说V'是一个规模为k的团

H3 归约的传递性

$$X \leq_P Y\text{且}Y \leq_P Z\text{,则}X \leq_P Z$$

H3 Complexity Classes

复杂度类关注的是随着输入大小n的增加，资源需求的增长率。所讨论的资源可以是时间(本质上是抽象机器上原始操作的数量)，也可以是(存储)空间。

一种抽象的度量，并不以秒或字节为单位给出时间或空间的要求。

根据已知最快的算法为决策问题分配复杂性类别。因此，决策问题可能会改变类，如果更快的算法被发现。

P类

P类问题的判别形式可以在多项式时间内解决

NP类

NP类问题可以在多项式时间内验证

P是NP的真子集

N=NP?

换句话说，解决问题和验证解决方案一样容易吗?
如果P=NP，那么对于一个YES实例，一个有效的“验证”解决方案意味着一个有效的“发现”解决方案

SAT，VERTEx覆盖，汉密尔顿圈等NP问题将会有有效的算法

若P≠NP:这类问题没有有效的算法。

NPC

如果所有的NP类问题都能归约到问题Y，则问题Y就是NP-Complete问题

之前提到的 3SAT Clique 顶点覆盖问题都是NPC问题

如何证明一个问题是NPC问题

集合覆盖问题

Input:一个集合U={e1,e2,...,en}和它的子集的集合S={S1,S2,...,Sm}和一个整数k

Output：是否存在S的一个子集C，使得C中所有子集并起来等于U，且|C|$\leq k$

证明过程：

01. 证明集合覆盖问题是NP问题
02. 选择一个NPC问题，顶点覆盖
03. 证明顶点覆盖问题多项式时间内可以归约到集合覆盖问题

定理

$$\text{顶点覆盖问题} \leq_P \text{集合覆盖问题}$$

给出一个顶点覆盖实例<G=(V,E),k>,可以通过以下过程构造一个集合覆盖实例<U,S,k>

01. U=E
02. 对于V中的每个顶点，构造一个集合S_v={e∈E:e与v是关联的}

我们可以断言

$G=(V,E)$有一个规模为k的集合覆盖 $\Longleftrightarrow$ (U,S)有一个规模为k的顶点覆盖

$\Rightarrow$:

让V的真子集X是G中一个规模为k的集合覆盖，Y={S_v:v∈X}是规模为k的集合覆盖

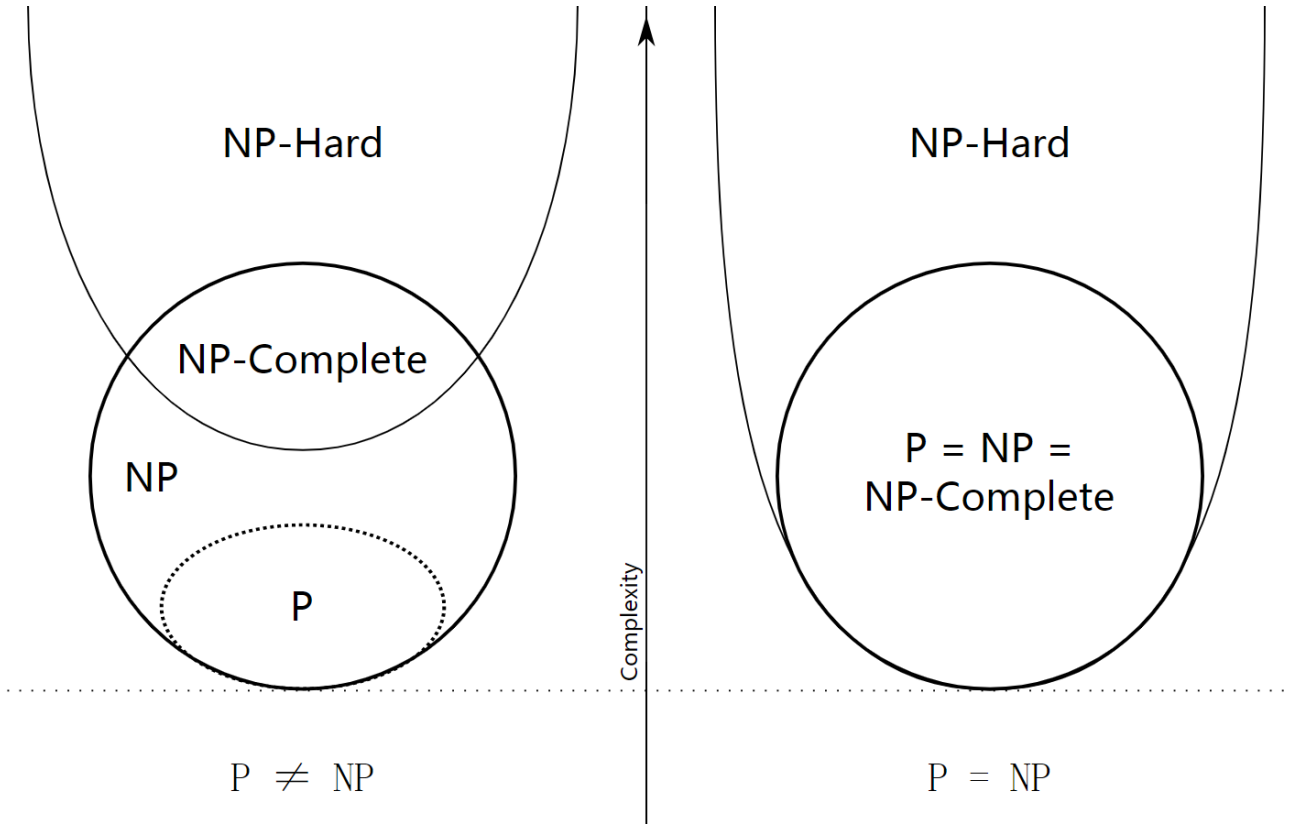
⇐:

S的真子集Y为规模为k的集合覆盖

X=$\{v:S_v \in Y\}$ 是G中一个规模为k的顶点覆盖

NP-hard

所有的NPC问题都能归约到问题Y，则该问题就是NP-hard问题



如何解决困难问题？ 做出质量和时间之间的取舍

- 放弃最优约束(近似算法)
- 放弃多项式时间约束(精确算法)
- 放弃确定性约束(随机算法)

H3 近似算法的概念

一个优化问题的α-近似算法需要满足

- 算法在多项式时间内运行
- 算法输出的值S在最优值OPT的一个因子α之内

$$Value(S) \leq \alpha OPT(\text{最小化问题})$$
$$Value(S) \geq \alpha OPT(\text{最大化问题})$$

α：算法的近似比或近似因子

- 对于最小化问题α>1
- 对于最大化问题α<1

H3 近似算法设计的关键

近似算法设计的困境

为了保证解的质量，我们必须将解和最优值进行比较，但是求得最优值一个是NP-hard问题

没有OPT的确切值，我们如何与OPT建立联系呢?

找到OPT的下界，与下界比较解的质量

01. 找到OPT的严格下界
02. 设计一个可行解的求解流程并将可行解与OPT的下界进行比较

近似算法举例

顶点覆盖问题

用一个相似的问题找到问题的最优解的下界

用最大匹配问题近似顶点覆盖问题

```
Vertex-Cover-Appro-With-MM algorithm
find a maximal matching M in G
S← {u∈V:存在(u,v)∈M}
return S
```

引理1 对于G中任何匹配M和任意顶点覆盖问题S

$$|S| \geq |M|$$

证明：

观察到对于匹配中的每条边，我们需要在顶点覆盖中包含至少1个它的端点，以便覆盖这条特定的边。因此S≥M

推论：

$$min_{S\ is\ a\ vertex\ cover}|S| \geq max_{M\ is\ a\ maximal\ matching}|M|$$

引理2 S是一个顶点覆盖

反证法

定理 Vertex-Cover-Appro-With-MM algorithm 是一个顶点覆盖算法的2-近似算法

证明：

$$|S| = 2|M| \leq 2OPT$$

H4 旅行商问题

Input：完全无向图G=(V,E)，每条边上有一个非负整数cost(u,v)

Output：一个具有最小代价的哈密尔顿回路

旅行商问题与三角不等式

Input：完全无向图G=(V,E)，每条边上有一个非负整数c(u,v)且对于所有顶点u,v,w∈V

$$c(u,v) \leq c(u,w) + c(w,v)$$

Output：一个具有最小代价的哈密尔顿回路

```
Approx-TSP-Tour(G,c)
选择G的顶点集V中的一个顶点r作为根
使用MST-Prim(G,c,r)计算G的最小生成树T
让H为一个顶点序列，根据它们在T中前序遍历的顺序进行排序
return H
```

定理：

APPROX-TSP-TOUR时一个用于解决满足三角不等式的旅行商问题的多项式时间2近似算法

证明：

设H*表示给定顶点集合上的一个最优旅行路线，我们通过删除一个旅行路线上的任一一条边得到生成树，且每条边的代价都是非负的，因此，最小生成树T的权值时最有旅行路线代价的一个下限

$$c(T) \leq c(H^*)$$

对T进行遍历时，访问节点的次序为W，W恰好经过T的每条边两次

$$c(W) = 2c(T)$$

进而

$$c(W) \leq 2c(H^*)$$

因为满足三角不等式，所以从W中删除重复的顶点，不会增加权值，删除重复顶点后得到的恰好是T的前序遍历序列，记作H

$$c(H) \leq c(W) \leq 2c(H^*)$$
$$c(H) \leq 2c(H^*)$$