数据结构与算法

DATA STRUCTURE

第二十三讲 最短路径与动态规划

胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

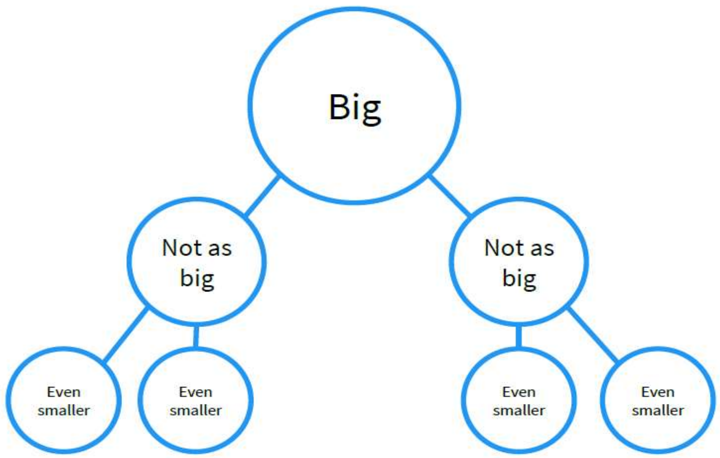
课堂内容

- 动态规划
- 最短路径算法

动态规划（DP）
Dynamic Programming

分而治之法

- 分治法 **Divide and Conquer**
  - 分：分成较小的可以递归解决的问题
  - 治：从子问题的解形成原问题的解



- 在分而治之法中，递归是“分”，“治”需要额外的开销
- 遇到算法难题，先套用分治法试试

动态规划思想

用来求一类问题的最优解（算法中最重要的方法之一）

- 有最优子结构：
  - 原问题的最优解可以由子问题的最优解推导出

- 原问题可分成有很多重叠的子问题
  - 类似分治法，区别是这些子问题有重叠
  - 排序方法中，子问题没重叠，用分治法
  - 计算Fibonacci数列，子问题重叠，用DP

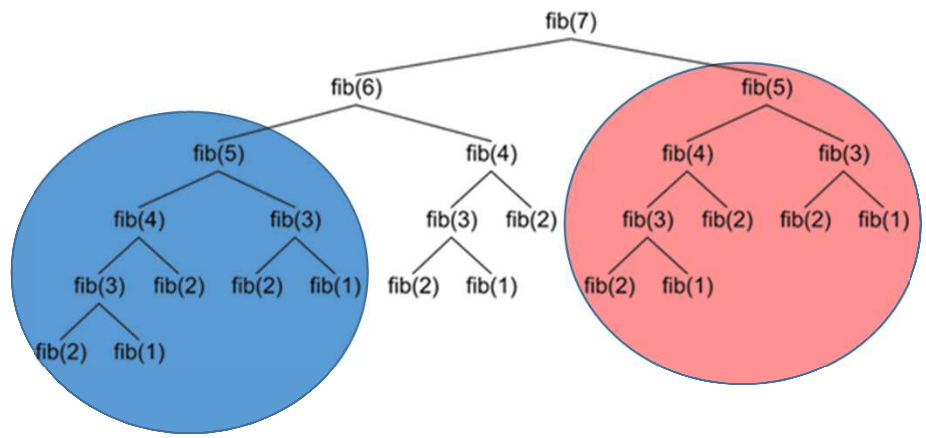
求解DP方式

- 把原问题描述/转换成递归表达式：
  - 递归+记忆的方式
    - 从顶向下（top down）
    - 求解需要的子问题
  - 迭代+建表格的方式
    - 从下往上（bottom up）
    - 求解所有子问题

Fibonacci函数的递归实现

$$Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2),$$
$$Fibonacci(1) = Fibonacci(2) = 1$$

```
long long Fibonacci(int n)
{
    if (n <= 2)
    {
        return 1;
    }
    return Fibonacci(n-1) + Fibonacci(n-2);
}
```



计算Fibonacci(50)

```
12586269025

Process returned 0 (0x0)   execution time : 60.107 s
```

问题是需要递归两个子问题，而且是重叠的

Fibonacci Top down

- 类似于递归
- 区别是先查表，看有没有已经知道的结果
- 要没有结果，再计算后存入查找表，以便之后使用

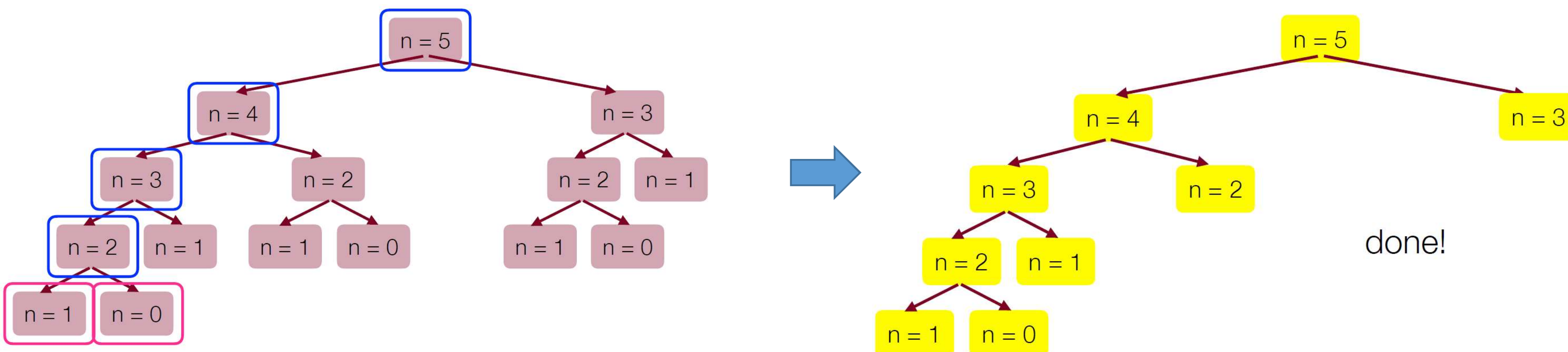
```
long long g_fibTable[100] = {};
long long Fibonacci_Topdown(int n)
{
    if (g_fibTable[n] > 0)
    {
        return g_fibTable[n];
    }

    if (n <= 1)
    {
        g_fibTable[n] = n;
        return n;
    }

    g_fibTable[n] = Fibonacci_Topdown(n-1) + Fibonacci_Topdown(n-2);

    return g_fibTable[n];
}
```

- 如果不是所有子问题都需要有解的话，用topdown方法可能不错

```
int main(void)
{
    cout << Fibonacci_Topdown(90) << endl;
}
```

```
2880067194370816120

Process returned 0 (0x0)   execution time : 0.024 s
```

记忆子问题结果（topdown）



Fibonacci bottom up

- 需要保存所有子问题的解
- 从最小问题开始求解了所有子问题

- 如果所有的子问题至少求解一遍的话，用bottomup方法比较好

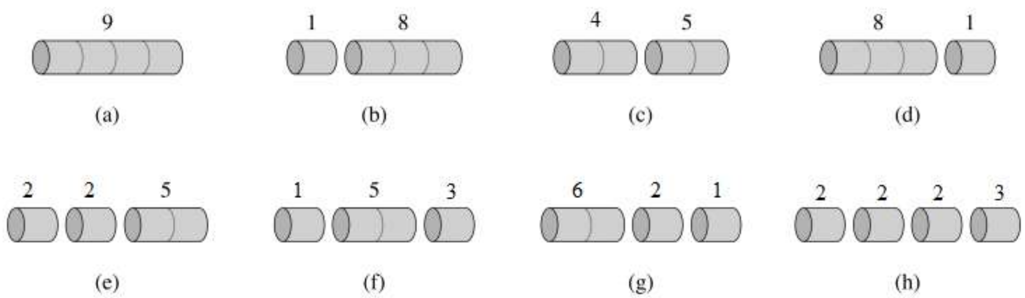
- 代码可能比topdown会复杂，因为topdown用了递归（系统帮助记忆顺序）

```
long long Fibonacci_Bottomup(int n)
{
  long long fib[100]{};
  fib[1] = 1;

  for (int i = 2; i <= n; i++)
  {
      fib[i] = fib[i-1] + fib[i-2];
  }

  return fib[n];
}


int main(void)
{
    cout << Fibonacci_Bottomup(90) << endl;
}
```

```
2880067194370816120

Process returned 0 (0x0)   execution time : 0.024 s
```

切割问题

length i	1	2	3	4	5	6	7	8	9
price p_i	1	5	8	9	10	17	17	20	21



- 把长度为n的金属杆子切分开，使得切割后价值和最大。长度和价格的关系见表格

递归

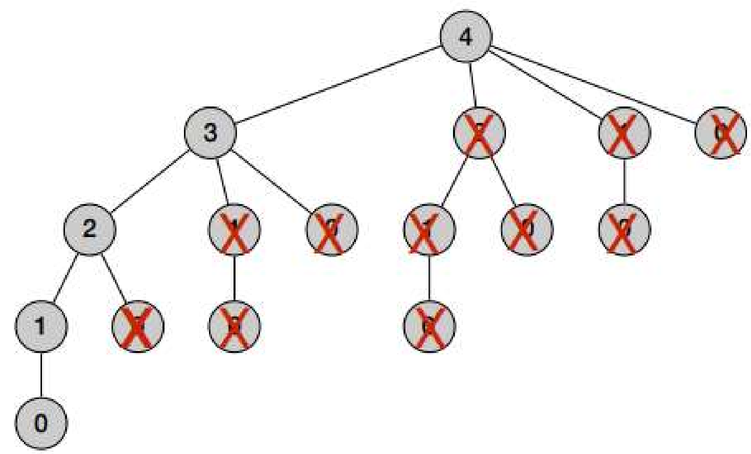
length i	1	2	3	4	5	6	7	8	9
price p_i	1	5	8	9	10	17	17	20	21

- 把问题写为递归的表达式
- 对长度为n的金属杆切割问题可以看作先切出第一段来（第一段长度1……n），然后对剩下长度的金属杆切割（子问题）

- $Rod(n) = \begin{cases} 0 & n = 0 \\ \max\limits_{i=1\ldots n}[p_i + Rod(n-i))] & n > 0 \end{cases}$

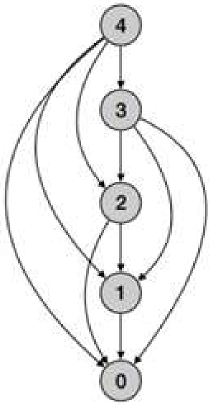
递归实现

```c
int Rod(int n)
{
    if (n <= 0)
    {
        return 0;
    }

    int maxValue = INT_MIN;
    for(int i = 0; i < n; i++)
    {
        if (maxValue < p[n-i] + Rod(i))
        {
            maxValue = p[n-i] + Rod(i);
        }
    }
    return maxValue;
}
```



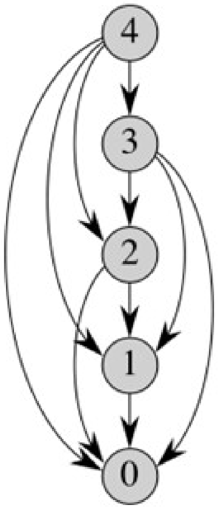
递归+记忆

```
int g_value[N]{};
int Rod(int n)
{
    if (g_value[n] > 0)
    {
        return g_value[n];
    }
    if (n <= 0)
    {
        g_value[0] = 0;
        return 0;
    }
    int maxValue = 0;
    for (int i = 1; i < n; i++)
    {
        if (maxValue < p[n-i] + Rod(i))
        {
            maxValue = p[n-i] + Rod(i);
        }
    }
    g_value[n] = maxValue;
    return maxValue;
}
```



迭代+表格

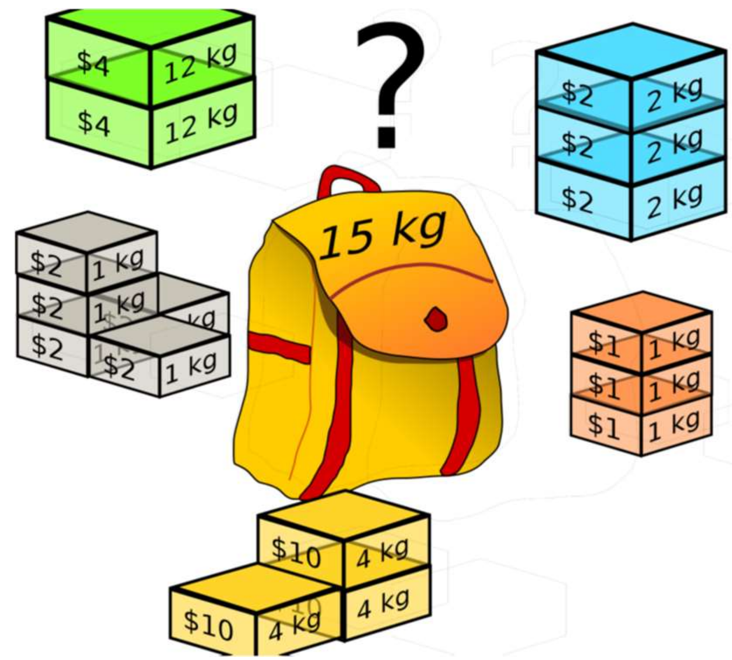
```
int Rod(int n)
{
    int value[N]{};

    for (int j = 1; j <= n; j++)
    {

        int maxValue = 0;
        for (int i = 0; i < j; i++)
        {
            if (maxValue < p[n-i] + value[i])
            {
                maxValue = p[n-i] + value[i];
            }
        }
        value[j] = maxValue;
    }
    return value[n];
}
```



Knapsack问题

- 有个背包最多装X容积，有几种物品，体积为w_i，价值为v_i，每种物品可以重复选。问最多能装多少价值的物品？

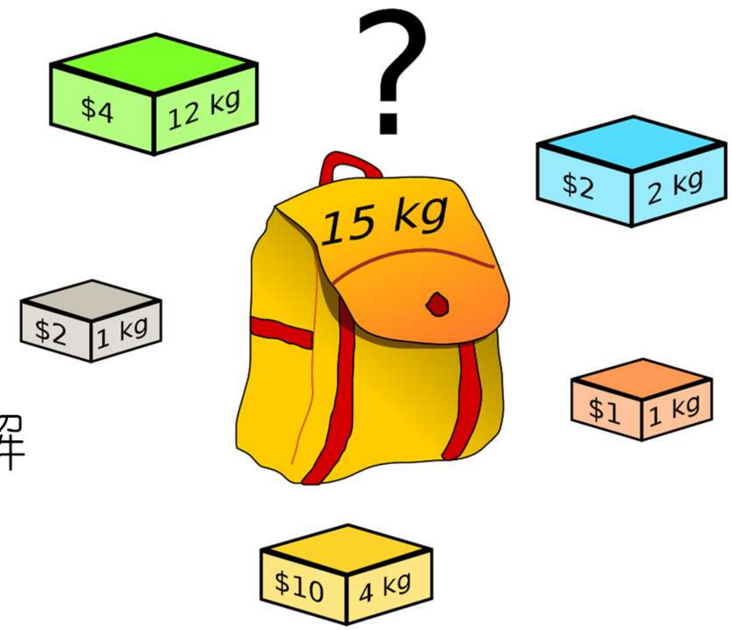
- K[x]是容积是x的最优解
- 类似金属杆切割问题的递归表达式



$$K[x] = \begin{cases} 0 & \text{if there are no } i \text{ where } w_i \le x \\ \max_i\{K[x-w_i] + v_i\} & \text{otherwise} \end{cases}$$

0/1 Knapsack问题

- 进一步，如果我们限制每种物品只有一个的话
- 先给物品分个序号，
- K[x,j]是容积是x，拿最多前j个东西的最优解



$$K[x,j] = \begin{cases} 0 & \text{if } \mathbf{x} \text{ or } \mathbf{j} \text{ are } 0 \\ \max\{K[x,j-1],\ K[x-w_j,j-1] + v_j\} & \text{otherwise} \end{cases}$$

Longest Common SubSequence

- 给定两个字符串X和Y，长度分别为m，n，问他们的最长公共字串是什么？
- T(i, j)是X前缀[0,…,i]和Y前缀[0,…,j]的LCS

$$
T(i, j) = \begin{cases} 0 & \text{if } \mathbf{i} \text{ or } \mathbf{j} \text{ is -1} \\ 1 + T(i-1, j-1) & \text{if } X[i] = Y[j] \text{ and } \mathbf{i}, \mathbf{j} \geq 0 \\ \max\{T(i-1, j), \\ \quad T(i, j-1)\} & \text{if } X[i] \neq Y[j] \text{ and } \mathbf{i}, \mathbf{j} \geq 0 \end{cases}
$$

最短路径算法

Bellman-Ford

Bellman-Ford算法

- 计算从单源s出发到别的顶点的最短路径
- 可以处理负边
- 检测负环

Bellman-Ford算法

- 初始化

For each vertex u

$\quad \boldsymbol{d}(\boldsymbol{u}) = \infty$

$\boldsymbol{d}(\boldsymbol{s}) = 0$

- 最短距离迭代计算过程

For $i = 1, \ldots, n-1$:

$\quad$ **For** each edge (u, v) in E:

$\quad\quad \boldsymbol{d}(\boldsymbol{v}) \leftarrow \boldsymbol{min}\{\boldsymbol{d}(\boldsymbol{v}), \boldsymbol{d}(\boldsymbol{u}) + \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})\}$

- 检测负环

For each edge (u, v) in E:

$\quad$ **If** $\boldsymbol{d}(\boldsymbol{v}) > \boldsymbol{d}(\boldsymbol{u}) + \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})$

$\quad\quad$ **return** "negative cycle"

为什么正确

定理：Bellman-Ford算法结束后，所有顶点的预估值$d(v)$都是实际最短距离$\delta(s, v)$

证明：用数学归纳法证明

- 假设第i次迭代后，对于顶点v，如果其真正最短路径最多$\boldsymbol{i}$条边，那么$d(v) = \delta(s, v)$

- 首先，base case：第0次迭代后，$d(s) = 0 = \delta(s, s)$

用动态规划方式求解

如何分解原问题

- 原问题：从起点s出发到任意顶点u的最短路径$\delta(s,u)$
  - 不好找子问题，比如
  - 从s到u的最短路径$\delta(s,u)$是从s到v的的最短路径$\delta(s,v)$的子问题？

- 按顶点划分，比如子问题是最短路径，且最多只包含前k个顶点
  - BaseCase是最短路径不含任何别的顶点
  - 我们想要找最终的最短路径可以包含任何别的顶点
  - 分解子问题解决了
  - 但是合并成原问题的解的时候复杂度相对比较高

- 按边的数目划分这里更容易合并子问题
  - 参考之前证明算法正确性过程

子问题

- 对于任意顶点v，求$d^k(v)$是从源点s到v的真正最短路径，且包含最多k条边 （理解子问题定义）

- $k = 0$时是初始状态，即包含0条边的最短路径
- 我们想求$k = n - 1$时，$d^{n-1}(v)$的值（和原问题等价）

- 引入k之后，我们才能建立从初始状态到最终目的之间的关系
- 我们希望通过$d^{k-1}(v)$，求解$d^k(v)$

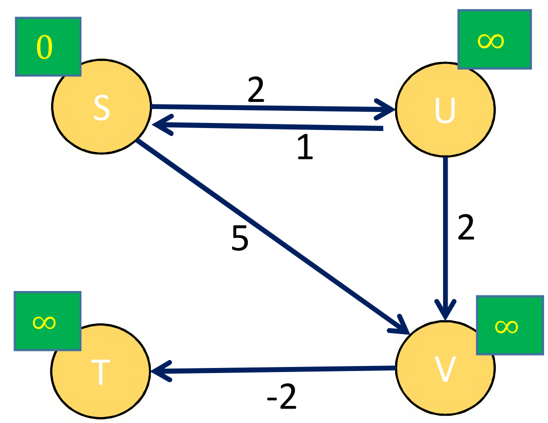
例子

- 子问题：求$d^k(v)$是从源点s到v的真正最短路径，且包含最多k条边

最多包含0条边最短路径

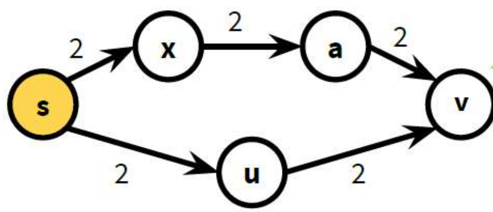
想通过$d^{(0)}$，计算$d^{(1)}$，即最多包含1条边最短路径

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$				
$d^{(2)}$				
$d^{(3)}$				



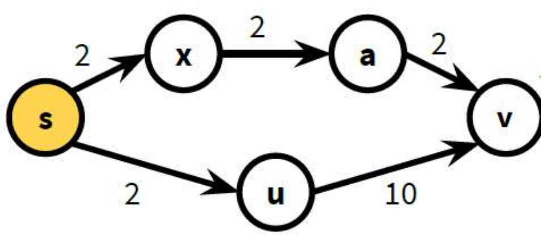
DP问题分解

- 子问题：求$d^k(v)$是从源点s到v的真正最短路径，且包含最多k条边
  - 想通过$d^{(k-1)}$，计算$d^{(k)}$，即最多包含k条边的最短路径
- Case 1：从s到v的真正最短路径包含最多$k-1$条边



这里$k=3$,
$d^k(v) = d^{k\text{-}1}(v)$

- Case 2：从s到v的真正最短路径包含k条边



这里$k=3$,
$d^k(v) = d^{k\text{-}1}(a) + w(a,v)$

递归表达式

- 子问题：求$d^k(v)$是从源点s到v的最短路径，且包含最多k条边
- 那么递归表达式是

$$d^k(v) = \begin{cases} \infty & k = 0, v \neq s \\ 0 & k = 0, v = s \\ \min\{d^{k-1}(v),\ \ \min\limits_{u}\{d^{k-1}(u) + w(u,v)\}\} & \end{cases}$$

Case 1

Case 2：对所有的顶点u，存在边$\{u, v\}$

Bellman-Ford算法递归实现

Double BF(**Node** v, **int** k)

  If k == 0

    return ∞ or 0

  Double $minD$= $\boldsymbol{BF(v, k{-}1)}$

  For each edge (u, v) in E:

    $minD = \boldsymbol{min}\{minD, \boldsymbol{BF(u, k{-}1)} + \boldsymbol{w(u, v)}\}$

  return $minD$

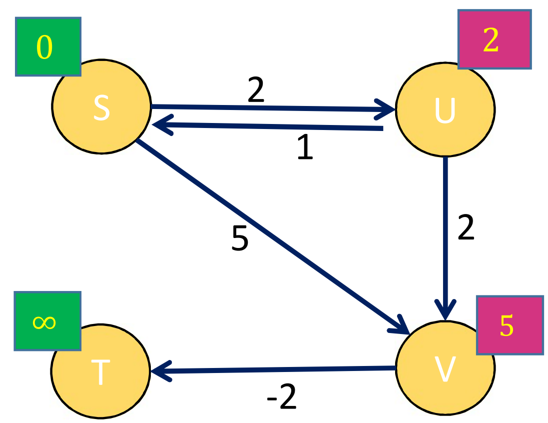
把递归表达式用递归方式实现，很简单。但是，
复杂度很高，因为大量的子问题在递归中不断重算
用记忆的方式来优化

例子

- 子问题：求$d^k(v)$是从源点s到v的最短路径，且包含最多k条边

记忆$d^{(0)}$[]，计算$d^{(1)}$，即最多包含1条边最短路径

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$				
$d^{(3)}$				

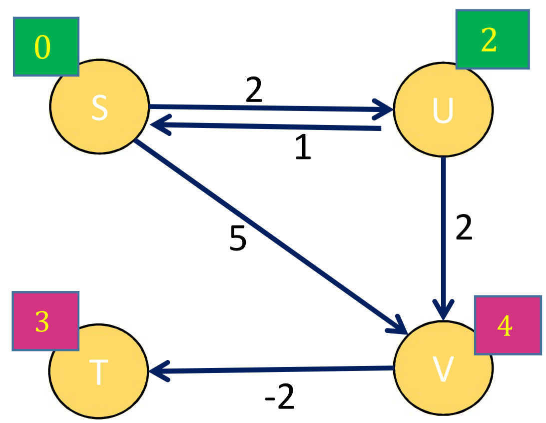


例子

- 子问题：求$d^k(v)$是从源点s到v的最短路径，且包含最多k条边

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$				

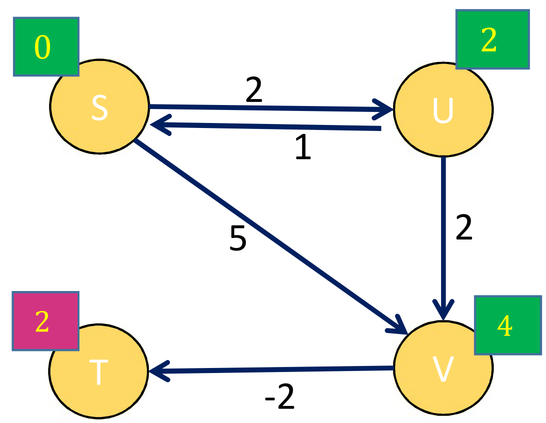
记忆$d^{(1)}$ []，计算$d^{(2)}$，即最多包含2条边最短路径



例子

- 子问题：求$d^k(v)$是从源点s到v的最短路径，且包含最多k条边

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2



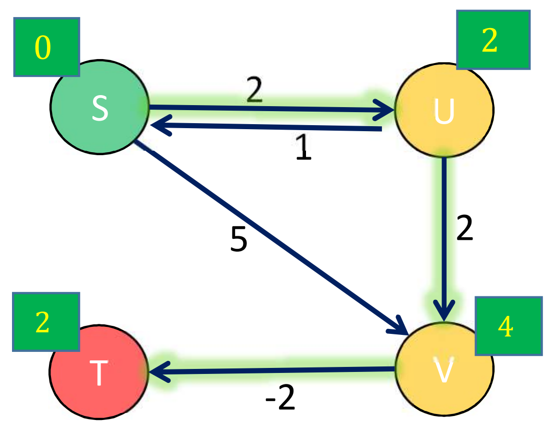
通过$d^{(2)}$ []，计算$d^{(3)}$，即最多包含3条边最短路径

例子

- 子问题：求$d^k(t)$是从源点s到t的最短路径，且包含最多k条边
  - 从$d^0(t) = \infty$，即从s到t的最多包含0条边最短路径没有
  - 从$d^1(t) = \infty$，即从s到t的最多包含1条边最短路径没有
  - 从$d^2(t) = 3$，即从s到t的最多包含2条边最短路径值是3
  - 从$d^3(t) = 2$，即从s到t的最多包含3条边最短路径值是2

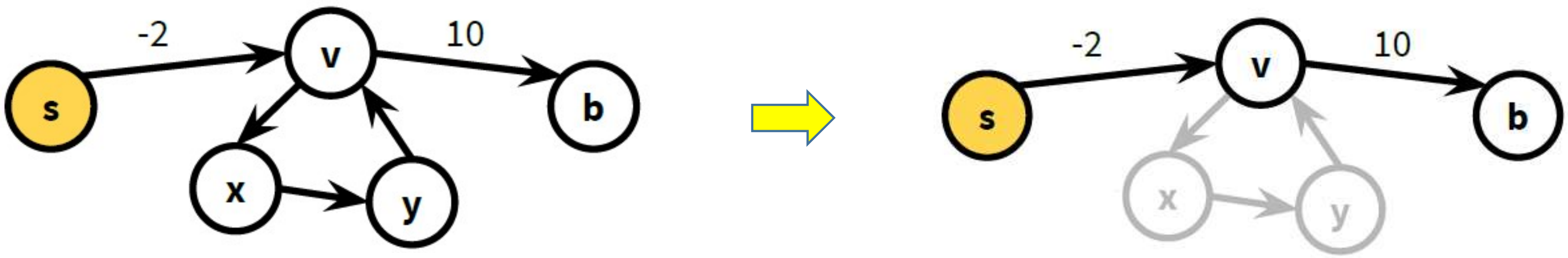
注意空间上来说，只需要保存相邻两组$d^{(k-1)}$ [] 和 $d^{(k)}$ []

	s	u	v	t
$d^{(0)}$	0	∞	∞	∞
$d^{(1)}$	0	2	5	∞
$d^{(2)}$	0	2	4	3
$d^{(3)}$	0	2	4	2

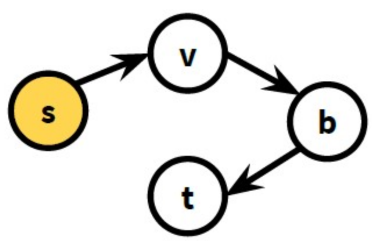


为什么正确

1) 没有负环的图上的最短路径最多只有$n-1$条边
   - 首先最短路径没有环cycle，既是简单路径



   - 其次，n个顶点的图上的简单路径最多$n-1$条边



为什么正确

2) 而且$d^{n-1}(v)$就是从源点s到v的包含最多$n-1$条边最短路径

$$d^k(v) = \begin{cases} \infty & k=0, v \neq s \\ 0 & k=0, v=s \\ \min\{\underbrace{d^{k-1}(v)}_{\text{Case 1}},\ \underbrace{\min_u\{d^{k-1}(u)+w(u,v)\}}_{\text{Case 2}}\} & \end{cases}$$

Case 1

Case 2：对所有的顶点u，存在边$\{u, v\}$

所以，n个顶点的图上的从源点s出发的最短路径等价于$d^{n-1}(v)$

Bellman-Ford算法topdown

- 注意在所需空间上
  - 可以把所有的$\boldsymbol{d}^k$()保存在表格(二维数组)
  - 也可以优化成对每个顶点$\boldsymbol{v}$，保存最新的$\boldsymbol{d}^k(\boldsymbol{v})$

时间复杂度

如果$|V| = n$， $|E| = m$，

- Bellman-Ford迭代算法复杂度是$O(mn)$

For $i = 1, \dots, n-1$:
  For each edge (u, v) in E:
    $\boldsymbol{d}(\boldsymbol{v}) \leftarrow \boldsymbol{min}\{\boldsymbol{d}(\boldsymbol{v}), \boldsymbol{d}(\boldsymbol{u}) + \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})\}$

- Bellman-Ford的DP算法复杂度同样是$O(mn)$（为什么）

先查表看$\boldsymbol{v}, k$对应的结果
Double $minD$= $\boldsymbol{BF}(\boldsymbol{v}, k-1)$
For each edge (u, v) in E:
  $minD = \boldsymbol{min}\{minD, \boldsymbol{BF}(\boldsymbol{u}, k-1) + \boldsymbol{w}(\boldsymbol{u}, \boldsymbol{v})\}$

Floyd-Warshall算法

计算两两最短路径

Floyd-Warshall算法

- All-Pairs Shortest Path (**APSP**)问题：任意两点之间的最短距离
- 另一个可以用DP方法求最优解的例子

- 直观思路
  - 对每个顶点，运行一遍Bellman-Ford算法
  - 也算DP求解，但时间复杂度$O(n^2m)$

- 有没有更优的方法？

如何分解原问题

- 原问题：从任意顶点u到任意顶点v的最短路径$\delta(u, v)$
  - 不好找子问题

- 按边的数目划分，比如子问题是最短路径，且最多只包含k条边
  - 求解的复杂度并不比上一页直观思路低
  - 同样需要加另一顶点的维度

- 按顶点划分这时候合适
  - 比按边划分原问题优
  - 当然比单源问题的按边划分复杂度高

定义子问题

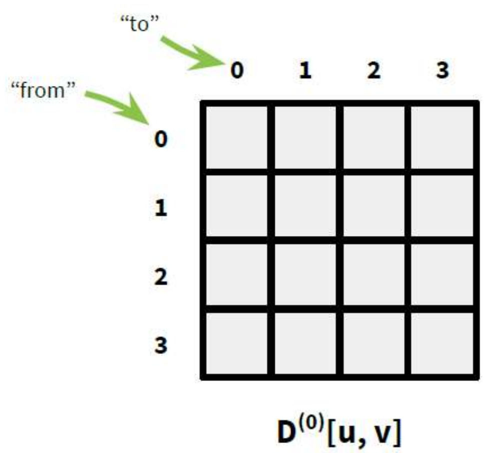
- 假如对所有顶点编号：$\{0,1,\ldots,n-1\}$
- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且路径内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$

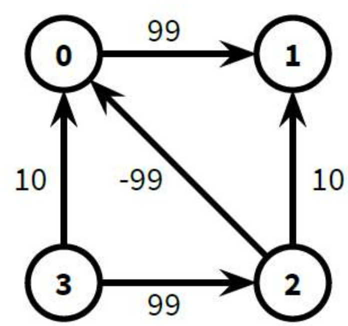
- 那么$k=0$时是初始状态，即最短路径不包含任何内部顶点
- 我们想求$k=n$时，$D^n(u,v)$的值（和原问题等价）

- 我们希望通过$D^{k-1}(u,v)$，求解$D^k(u,v)$

例子：初始解

- 假如对所有顶点编号：$\{0,1,\ldots,n-1\}$
- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且路径内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$





例子：初始解

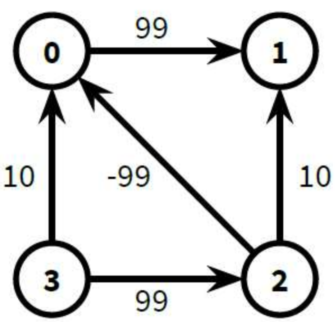
- 假如对所有顶点编号：$\{0,1,\ldots,n-1\}$
- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$

"to"

"from"

	0	1	2	3
0	0	99		
1		0		
2			0	
3	10			0

$D^{(0)}[u,v]$



例子：初始解

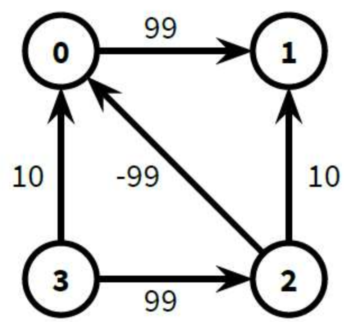
- 假如对所有顶点编号：$\{0,1,\ldots,n-1\}$
- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$

"to"

"from"

	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u,v]$



$D^{(0)}$是初始状态，即内部没有顶点的最短路径，其实就是直接有边的邻接顶点间可设最短距离

例子：归纳过程

- 假如对所有顶点编号：$\{0,1,\ldots,n-1\}$
- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$

"to"

"from"

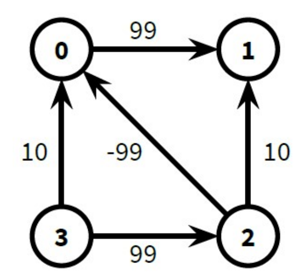
	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0				
1				
2		0		
3		109		

$D^{(1)}[u, v]$

通过$D^{(0)}$，计算$D^{(1)}$，即内部最多通过顶点0的最短路径



例子：归纳过程

- 假如对所有顶点编号：$\{0,1,\ldots,n-1\}$
- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$

“to”

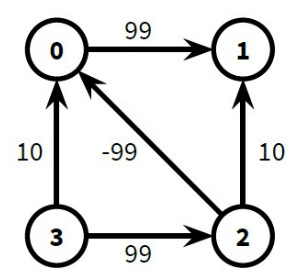
“from”

	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	10	0	∞
3	10	∞	99	0

$D^{(0)}[u, v]$

	0	1	2	3
0	0	99	∞	∞
1	∞	0	∞	∞
2	-99	0	0	∞
3	10	109	99	0

$D^{(1)}[u, v]$

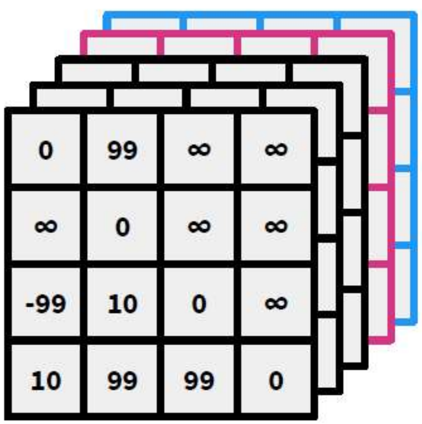


通过$D^{(0)}$，计算$D^{(1)}$，即内部最多通过顶点0的最短路径

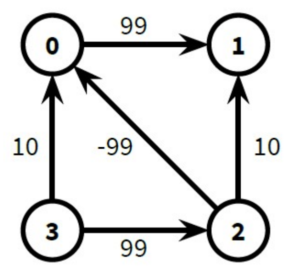
例子：归纳过程

- 假如对所有顶点编号：$\{0,1,\ldots,n-1\}$
- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$

o(|V|³)



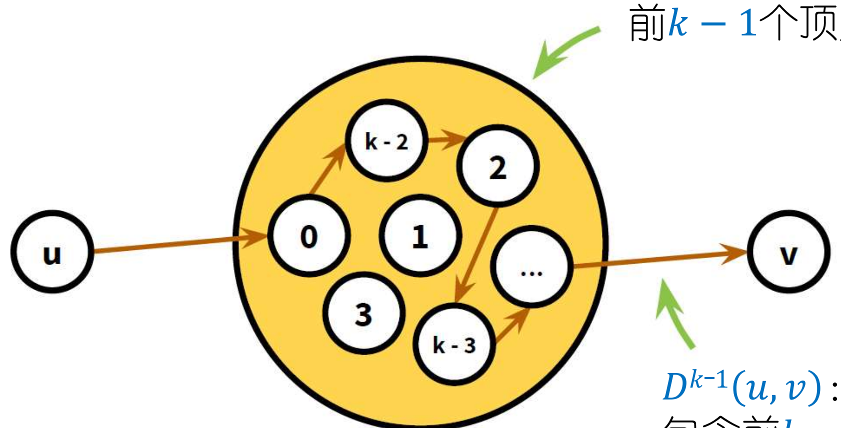
$D^{(4)}[u,v]$



DP问题分解

- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\dots,k-1$
  - 想通过$D^{k-1}(u,v)$，计算$D^k(u,v)$，即最多包含前k个顶点的最短路径

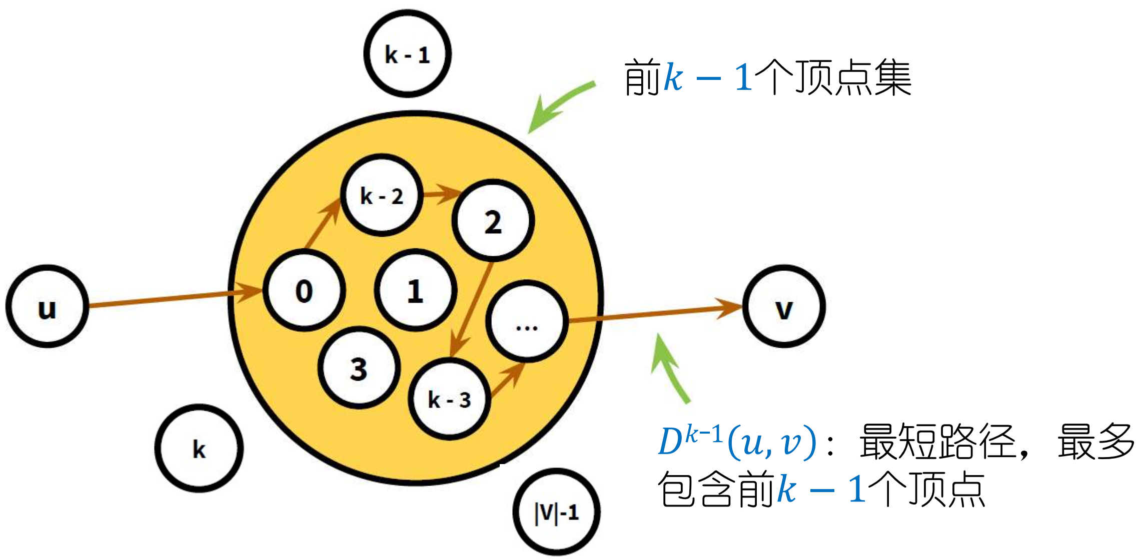
前$k-1$个顶点集



$D^{k-1}(u,v)$：最短路径，最多包含前$k-1$个顶点

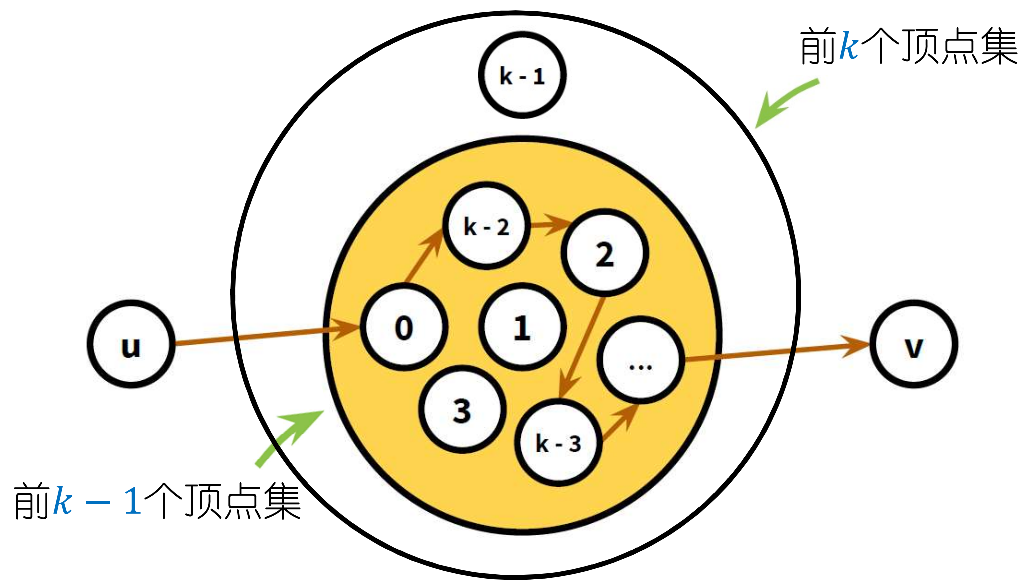
DP问题分解

- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$
  - 想通过$D^{k-1}(u,v)$，计算$D^k(u,v)$，即最多包含前k个顶点的最短路径



DP问题分解

- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$
  - 想通过$D^{k-1}(u,v)$，计算$D^k(u,v)$，即最多包含前k个顶点的最短路径

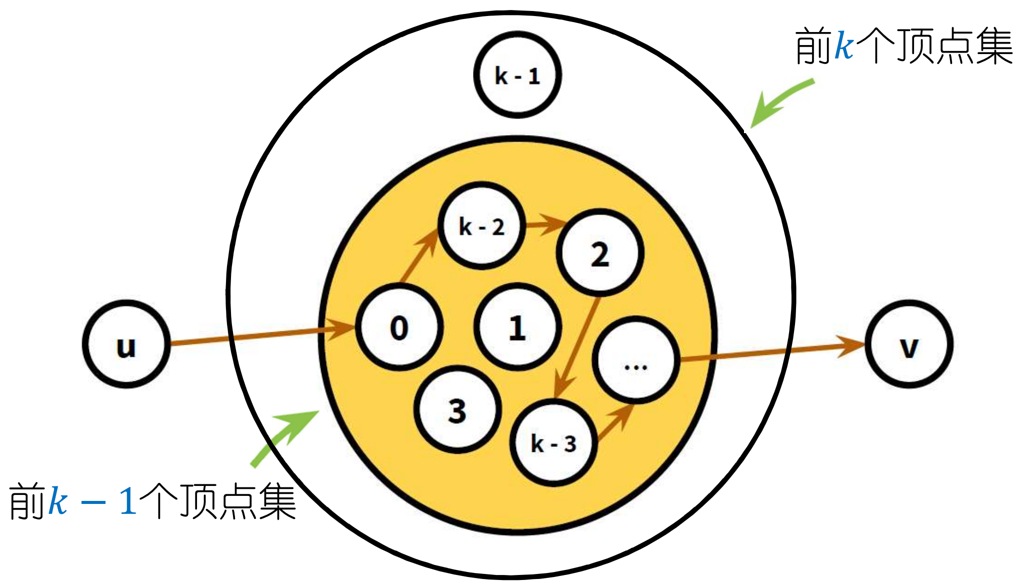


Case 1

想通过$D^{k-1}(u,v)$，计算$D^{k}(u,v)$，即最多包含前k个顶点的最短路径

- **Case 1**：从点u到v的真正最短路径$D^{k}(u,v)$，不需要通过顶点k-1，那么

$$\boldsymbol{D^{k}(u,v) = D^{k-1}(u,v)}$$



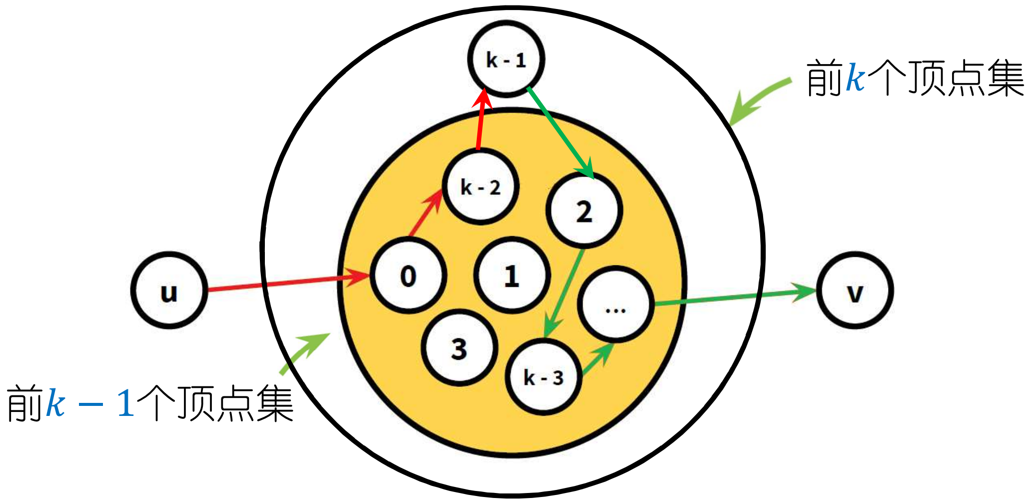
Case 2

想通过$D^{k-1}(u,v)$，计算$D^{k}(u,v)$，即最多包含前k个顶点的最短路径

Case 2：从点u到v的真正最短路径$D^{k}(u,v)$，需要通过顶点k-1，

- 路径上顶点k-1只出现一次，
- 分解为u到‘k-1’的最短路径，和‘k-1’到v的最短路径
- 都只包含前$k-1$个顶点集，

$$\boldsymbol{D^{k}(u,v) = D^{k-1}(u,k-1) + D^{k-1}(k-1,v)}$$



递归表达式

- 假如对所有顶点编号：$\{0,1,\ldots,n-1\}$
- 子问题：求$D^k(u,v)$是从点u到v的真正最短路径，且内部的顶点只用了最多前k个，即$0,1,\ldots,k-1$
- 那么递归表达式是

$$D^k(u,v)=\begin{cases} w(u,v) & k=0 \\ \min\{\underbrace{D^{k-1}(u,v)}_{\text{Case 1}},\ \underbrace{D^{k-1}(u,k-1)+D^{k-1}(k-1,v)}_{\text{Case 2}}\} & \end{cases}$$

Floyd-Warshall算法递归实现

Double FW(**Node** u, **Node** v, **int** k)

  If $k == 0$

    return $w(u, v)$

  return $\boldsymbol{min}\{\boldsymbol{FW}(\boldsymbol{u}, \boldsymbol{v}, k-1), \boldsymbol{FW}(\boldsymbol{u}, v_{k-1}, k-1) + \boldsymbol{FW}(v_{k-1}, \boldsymbol{v}, k-1)\}$

把递归表达式用递归方式实现，很简单。但是，
复杂度很高，因为大量的子问题在递归中不断重算
用记忆的方式来优化

Floyd-Warshall算法bottom-up

- 初始化

For each vertex i, j

$\quad \boldsymbol{D}^0(\boldsymbol{i}, \boldsymbol{j}) = w(i, j)$

For each vertex i, j and $k > 0$

$\quad \boldsymbol{D}^k(\boldsymbol{i}, \boldsymbol{j}) = \infty\ or\ 0 \text{ if } i = j$

- 最短距离迭代计算过程

For $k = 1, \ldots, n-1$:

$\quad$ **For** $i = 0, \ldots, n-1$:

$\quad\quad$ **For** $j = 0, \ldots, n-1$:

$$\boldsymbol{D}^k(\boldsymbol{i}, \boldsymbol{j}) \leftarrow \min\{\boldsymbol{D}^{k\text{-}1}(\boldsymbol{i}, \boldsymbol{j}),\ \boldsymbol{D}^{k\text{-}1}(\boldsymbol{i}, \boldsymbol{k}-\mathbf{1}) + \boldsymbol{D}^{k\text{-}1}(\boldsymbol{k}-\mathbf{1}, \boldsymbol{j})\}$$

因为k从小开始计算两两最短距离，循环中子问题都是查表结果

时间复杂度

如果$|V|=n$， $|E|=m$，

- Floyd-Warshall算法的DP实现复杂度是$O(n^3)$

For $k = 1, \dots, n-1$:
  For $i = 0, \dots, n-1$:
    For $j = 0, \dots, n-1$:
      $\boldsymbol{D}^k(\boldsymbol{i},\boldsymbol{j}) \leftarrow \min\{\boldsymbol{D}^{k-1}(\boldsymbol{i},\boldsymbol{j}),\ \ \boldsymbol{D}^{k-1}(\boldsymbol{i},\boldsymbol{k}-\mathbf{1}) + \boldsymbol{D}^{k-1}(\boldsymbol{k}-\mathbf{1},\boldsymbol{j})\}$

- 如果对于每个顶点简单执行Dijkstra算法，复杂度是

$$O(n^2\log(n)+mn)$$

检测负环

- Floyd-Warshall算法适用于负边的图
- 同样可以检测负环
- 只要看$D^n(u,u)$是不是有小于0的
  - 因为$D^0(u,u)$初始值为0
  - 如果存在负环，环上的任一顶点都会被更新到更小的值，即负值

小结

- 用DP求解问题的关键是第一步，即
  - 如何把原问题分解/转换成子问题
  - 写出递归表达式（问题分析比较偏向于数学）
- 实现DP过程时，相对简单。更多的是
  - 递归过程的理解，加上记忆中间结果
  - 或者递归到迭代的转换
- 更多DP例子：LCS、knapsack、 0/1 knapsack问题
- 三种最短路径算法比较

	Dijkstra	Bellman-Ford	Floyd-Warshall
Problem	Single source shortest path	Single source shortest path	All pairs shortest path
Runtime	$O(\|E\|+\|V\|\log(\|V\|))$ **worst-case** with a fibonacci heap	$O(\|V\|\,\|E\|)$ **worst-case**	$O(\|V\|^3)$ **worst case**

Q&A

Thanks!