第19章

动态规划

动态规划算法

- 与贪婪算法相同的是：
  - 将一个问题的解决方案视为一系列决策的结果。
- 与贪婪算法不同的是：
  - 在贪婪算法中，每采用一次贪婪准则便做出一个不可撤回的决策。
  - 在动态规划中，还要考察每个最优决策序列中是否包含一个最优子序列。

动态规划

- 动态规划方法采用最优原则（ principle of optimality)来建立用于计算最优解的递归式。
- 所谓最优原则即不管前面的策略如何，此后的决策必须是基于当前状态（由上一次决策产生）的最优决策。

动态规划法

- 在求解某些问题的时候，可以试着把问题分成必要多的子问题，每个子问题又可以分成数目不确定的必要多的子子问题，这样就会产生大量的子问题。如果分得的子问题界限不清，互相交叉，则在大量的子问题中会存在一些完全相同的子问题。
- 为了避免重复解这些相同的子问题，可以在解决一个子问题后把它的解保留下来，若遇到求解与之相同的子问题的时候，就可以把它找出来直接使用。
- 为解问题而将它的子问题的解填入表中以待需要时查表，这样的方法就是动态规划法。

0/1背包问题动态规划算法思想

- 0/1背包问题中：确定$x_1 \cdots\cdots x_n$的值
- 假设按i = 1，2，…，n 的次序来确定x_i 的值
  - x_1 = 0，则问题转变为相对于其余物品（即物品2，3，…，n），背包容量仍为c 的背包问题。
  - x_1 = 1，问题就变为关于最大背包容量为c-w_1 的问题
  - 设 $r \in \{c, c - w_1\}$ 为剩余的背包容量。
    - 在第一次决策之后，剩下的问题便是考虑背包容量为r 时的决策
    - 不管x_1 是0或是1，[x_2，…，x_n] 必须是第一次决策之后的一个最优方案

- 当最优决策序列中包含一个最优子序列时，可建立动态规划递归方程
- $f(i,y)$： 表示剩余容量为y，剩余物品为i，$i+1$，…，n 时的最优解的值，即：

$$f(n,y)=\begin{cases} p_n & y \geqslant w_n \\ 0 & 0 \leqslant y < w_n \end{cases}$$

$$f(i,y)=\begin{cases} \max\{f(i+1,y), f(i+1,y-w_i)+p_i\} & y \geqslant w_i \\ f(i+1,y) & 0 \leqslant y < w_i \end{cases}$$

- 0/1背包问题：求解$f(1,c)$

0/1背包问题动态规划算法

例19-4 n=3,w=[100,14,10],p=[20,18,15],c=116

若0≤y＜10， 则f(3,y)= 0；

若y≥10， f(3,y)=15。

利用递归式(15-2)得

$$f(n,y)=\begin{cases} p_n & y \geqslant w_n \\ 0 & 0 \leqslant y < w_n \end{cases} \qquad (19\text{-}1)$$

f (2,y) = 0 (0≤y＜10)

f(2,y)= 15（10≤y＜14）

f(2,y)= 18（14≤y＜24）

f(2,y)= 33（y≥24）。

$$f(i,y)=\begin{cases} \max\{f(i+1,y),f(i+1,y-w_i)+p_i\} & y \geqslant w_i \\ f(i+1,y) & 0 \leqslant y < w_i \end{cases} \qquad (19\text{-}2)$$

因此最优解

f(1,116)=max{f(2,116）,f(2,116-w1)+p1} =

max{f(2,116）,f(2,16)+20}=max{33,38}=38。

0/1背包问题动态规划算法

- 计算*xi* 值，步骤如下：
- 若f(1,c)=f(2,c)，则$x1$ =0，否则$x1$ =1。接下来需从剩余容量c-$w1$中寻求最优解，用
- f (2, c-$w1$)表示最优解。依此类推，可得到所有的xi (i= 1…n) 值。
- 在该例中，f (2,116)=33≠f (1,116)，所以$x1$ = 1。接着利用返回值38-$p1$=18 计算$x2$ 及$x3$，此时r = 116-$w1$ =16，又由f (2,16)=18，得f (3,16)=14≠f(2,16)，因此$x2$ =1，此时r=16-$w2$=2，所以f (3,2) =0，即得$x3$ = 0。

动态规划

- 编写一个简单的递归程序来求解动态规划递归方程，如果不努力地去避免重复计算，递归程序的复杂性将非常可观。

背包问题的递归函数

- int F(int i, int y)

```
{// 返回f ( i , y ) .
if (i == n) return (y < w[n]) ? 0 : p[n];
if (y < w[i]) return F(i+1,y);
return max(F(i+1,y), F(i+1,y-w[i]) + p[i]);
}
```

$$f(n,y)=\begin{cases} p_n & y \geqslant w_n \\ 0 & 0 \leqslant y < w_n \end{cases} \tag{19-1}$$

和

$$f(i,y)=\begin{cases} \max\{f(i+1,y), f(i+1,y-w_i)+p_i\} & y \geqslant w_i \\ f(i+1,y) & 0 \leqslant y < w_i \end{cases} \tag{19-2}$$

复杂性？

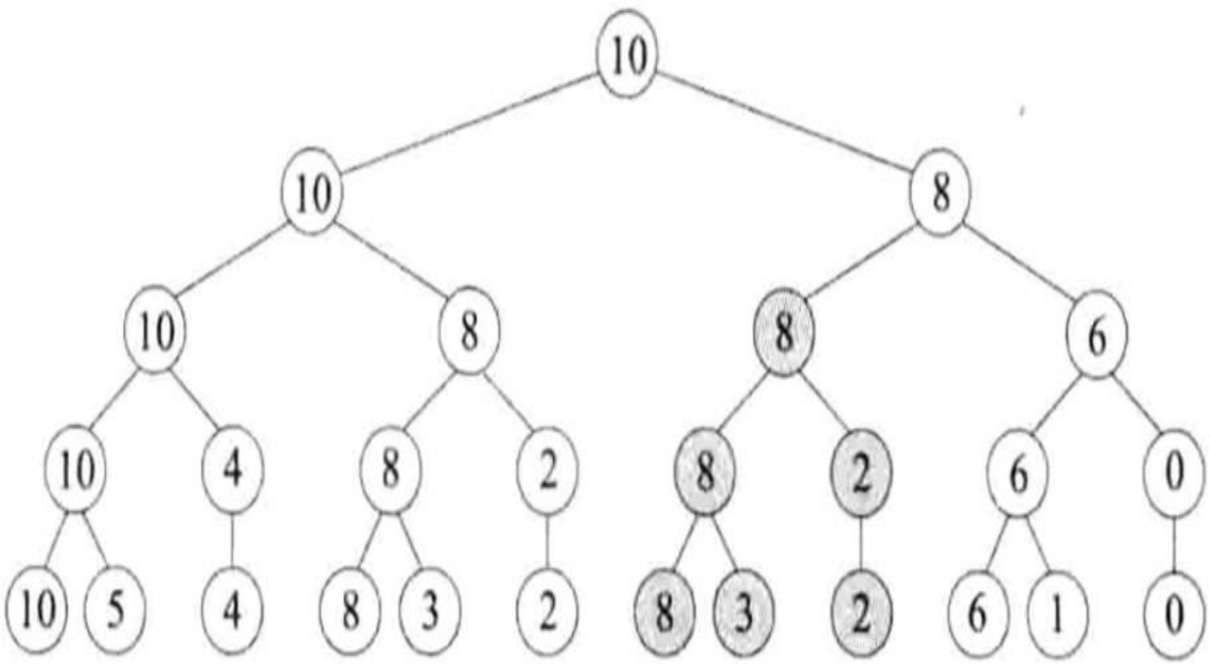
- t(1)=a;
- t(n)≤2t(n-1)+b(n>1)，其中a、b为常数。
- 通过求解可得t(n)=0(2^n)

$$f(n,y)=\begin{cases} p_n & y \geqslant w_n \\ 0 & 0 \leqslant y < w_n \end{cases} \tag{19-1}$$

和

$$f(i,y)=\begin{cases} \max\{f(i+1,y),f(i+1,y-w_i)+p_i\} & y \geqslant w_i \\ f(i+1,y) & 0 \leqslant y < w_i \end{cases} \tag{19-2}$$

例19-5 设n= 5，p= [6 , 3 , 5 , 4 , 6]，w=[2,2,6, 5,4] 且c= 1 0 ,求F (1 , 1 0)。递归调用的关系如图的树型结构所示。每个节点用y值来标记。对于第j层的节点有i=j，因此根节点表示F (1 , 1 0)，而它有左孩子和右孩子，分别对应F (2 , 1 0)和F (2 , 8)。



背包问题

2. 权为整数的迭代方法

当权为整数时，可设计一个相当简单的算法（见程序19 - 3）来求解*f* (1 ,*c*)。该算法基于例19-4所给出的策略，因此每个*f* (*i*,*y*) 只计算一次。程序19-3用二维数组f [ ][ ]来保存各*f* 的值。

而回溯函数Traceback用于确定由程序19 - 4所产生的x_i值。函数Knapsack的复杂性为O(nc)，而Tr a c e b a c k的复杂性为O(n)。

程序19-3　f 的迭代计算

```
void Knapsack(int p[], int w[], int c, int n, int** f)
{// 对于所有i和y计算f [ i ] [ y ]
// 初始化f [ n ] [ ]
yMax=min(w[n]-1,c);
for (int y = 0; y <= yMax; y++)
f[n][y] = 0;
for (int y = w[n]; y <= c; y++)
f[n][y] = p[n];
```

- // 计算剩下的f

程序19-3 f 的迭代计算

```
for (int i = n - 1; i > 1; i--) {
 yMax=min(w[i]-1,c);
  for (int y = 0; y <= yMax; y++)
     f[i][y] = f[i+1][y];
  for (int y = w[i]; y <= c; y++)
     f[i][y] = max(f[i+1][y], f[i+1][y-w[i]] + p[i]);
}
f[1][c] = f[2][c];
if (c >= w[1])
f[1][c] = max(f[1][c], f[2][c-w[1]] + p[1]);
}
```

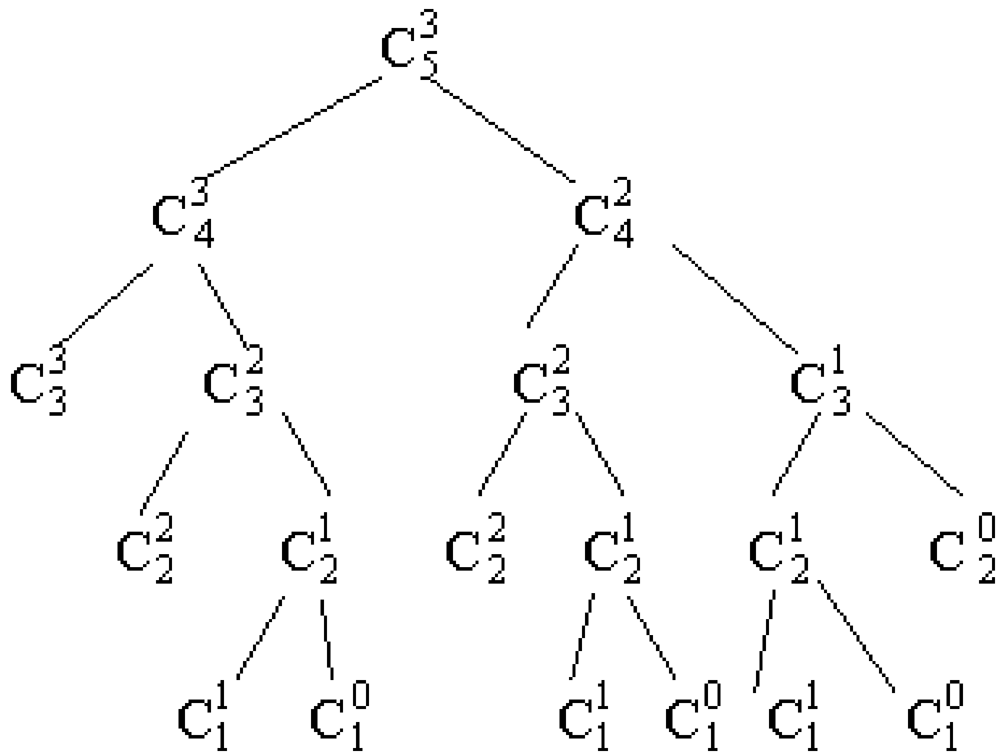
程序19-4 X 的迭代计算

```
template<class T>
void Traceback(T **f, int w[], int c, int n, int
   x[])
{// 计算x
for (int i = 1; i < n; i++)
if (f[i][c] == f[i+1][c]) x[i] = 0;
else {x[i] = 1;
c -= w[i];}
x[n] = (f[n][c]) ? 1 : 0;
}
```

程序19-3　f 的迭代计算

程序19-3有两个缺点：1) 要求权为整数；2) 当背包容量*c* 很大时，程序19 -3的速度慢于程序19 - 1。一般情况下，若$c > 2^n$，程序1 9 -3的复杂性为$\Omega(n2^n)$。

$$
\begin{cases}
C_m^n = C_{m-1}^n + C_{m-1}^{n-1}, & m > n > 0; \\
C_m^n = 1, & n = 0\text{或}m = n\text{。}
\end{cases}
$$

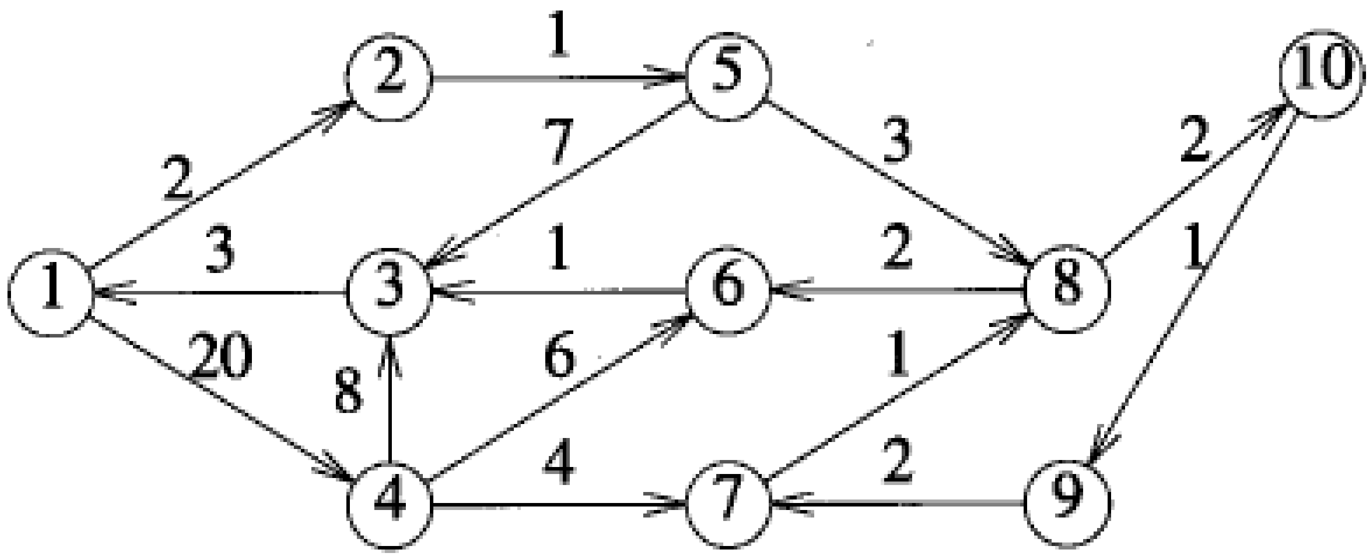


19.2.3 所有顶点对最短路径问题

- 问题：
  - 在n 个顶点的有向图G中，寻找**每一对顶点**之间的最短路径，即对于每对顶点(i, j)，需要寻找**从i到j 的最短路径**及**从j 到i 的最短路径，**对于无向图，这两条路径是一条。
- 对一个n 个顶点的图，需寻找p =$\mathbf{n(n\text{-}1)}$ **条最短路径**。

使用Dijkstra算法

- Dijkstra算法：边上的权值>=0
- 使用Dijkstra算法 n 次, 每次用1个顶点作为源点
- 时间复杂性： $O(n^3)$.



Floyd(弗洛伊德)最短路径算法

- 假定图*G*中不含有长度为负数的环路
- 设图*G*中*n* 个顶点的编号为1到*n*。
- 令**c(i,j,k)**表示从顶点***i*** 到顶点***j*** 的最短路径的长度，其中该路径中**允许经过的顶点都不大于*k*** 。

- $$c(i,j,0)=\begin{cases} \text{边 }(i,j)\text{的长度} & (i,j) \in E \\ 0 & i=j \\ +\infty \quad (\text{noEdge}) & \text{其它} \end{cases}$$

- C(i,j,0) = a[i][j] a 是耗费邻接矩阵

- c(i,j,n) 则是从顶点*i* 到顶点*j* 的最短路径的长度

- c(i,j,0) $\Rightarrow$ c(i,j,n) ？

i

- c(i,j,k-1)$\Rightarrow$ c(i,j,k), k > 0 ？

$c(i, j, k-1) \Rightarrow c(i, j, k)$， $k>0$

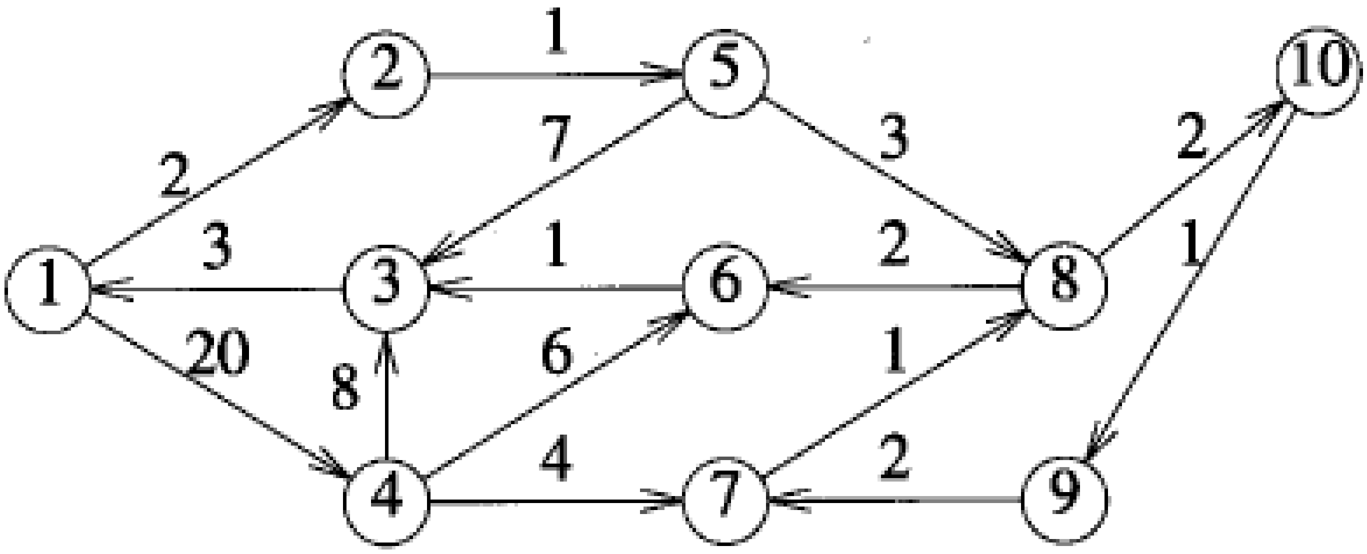
- $c(i,j,k)$ 有两种可能：
  - 1.该路径不含中间顶点*k*， 该路径长度为$\boldsymbol{c(i, j, k\text{-}1)}$
  - 2. 该路径含中间顶点*k*，路径长度为
  - $\boldsymbol{c(i, k, k\text{-}1) + c(k, j, k\text{-}1)}$。

- 结合以上两种情况，$c(i, j, k)$ 取两者中的最小值

  $$\Rightarrow c(i,j,k) = \min\{ c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\}.$$

- 按 $k = 1, 2, 3, \ldots, n$的顺序计算$c(i,j,k)$

- $c(1,3,k) = \infty$ k=0，1，2，3
- $c(1,3,4) = 28$
- $c(1,3,k) = 10$ k=5，6，7
- $c(1,3,k) = 9$ k=8，9，10



Floyd算法的伪代码

```
//寻找最短路径的长度
//初始化c(i，j，0)
for(int i=1;i<=n;i++)
for (int j=1;j<=n;j++)
    c(i,j,0)=a(i,j);//a是长度邻接矩阵
//计算c(i,j,k)(1≤k≤n)
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      if (c(i,k,k-1) + c(k,j,k-1)< c(i,j,k-1) )
            c(i,j,k) = c(i,k,k-1) + c(k,j,k-1)
       else     c(i,j,k) = c(i,j,k-1)
```

- 若用c(i,j) 代替c(i,j,k) ,最后所得的*c*(*i*,*j*) 之值将等于*c*(*i*,*j*,*n*) 值

计算最短路径

- 令kay(i,j) 表示从i 到j 的最短路径中最大的*k*值。
- 初始, kay(i,j)=0 (最短路径中没有中间顶点).

```
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      if (c(i,j) > c(i,k) + c(k,j))
       {kay(i,j) = k; c(i,j) = c(i,k) + c(k,j);}
```

AdjacencyWDigraph::Allpairs 1/2

```
template<class T>
void  Allpairs(T **c, int **kay)
{//所有点对的最短路径;对于所有i和j，计算c[i][j]和kay[i][j]
//初始化c[i][j]=c(i，j，0)
for (int i=1;i<=n;i++)
for (int j=1;j<=n;j++){
    c[i][j]=a[i][j];
    kay[i][j]=0;}
for  (i=1;i<=n;i++)
    c[i][i]=0;
```

AdjacencyWDigraph::Allpairs 1/2

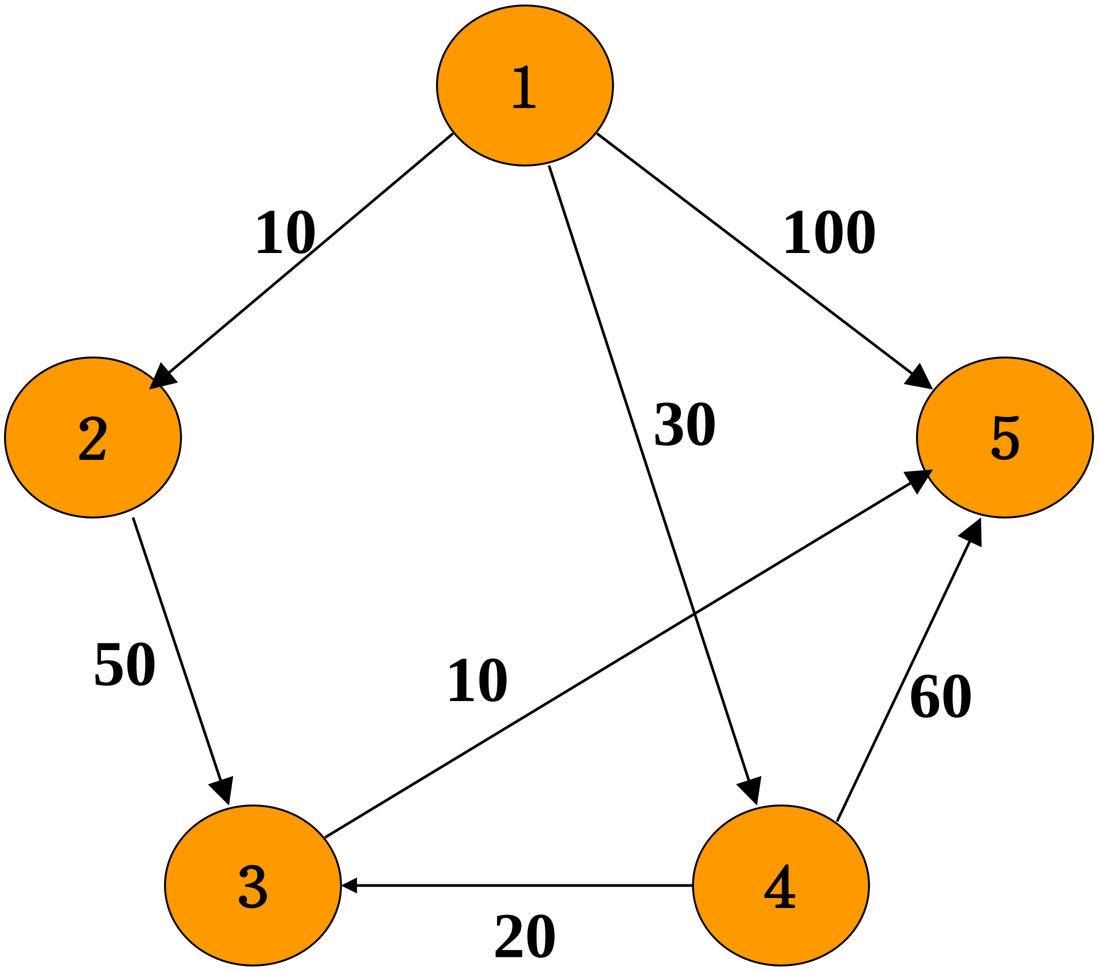
```
//计算c[i][j]=c(i,j,k)
for (int k=1;k<=n;k++)
for (int i=1;i<=n;i++)
for (int j=1;j<=n;j++)
    {if  (c[i][k]!=NoEdge && c[k][j]!=NoEdge &&
         (c[i][j]==NoEdge || c[i][j] > c[i][k] + c[k][j]))
         {c[i][j]= c[i][k] + c[k][j];
          kay[i][j]=k;
         }
    }
}
```

- 时间复杂性： $\Theta(n^3)$.

复杂性

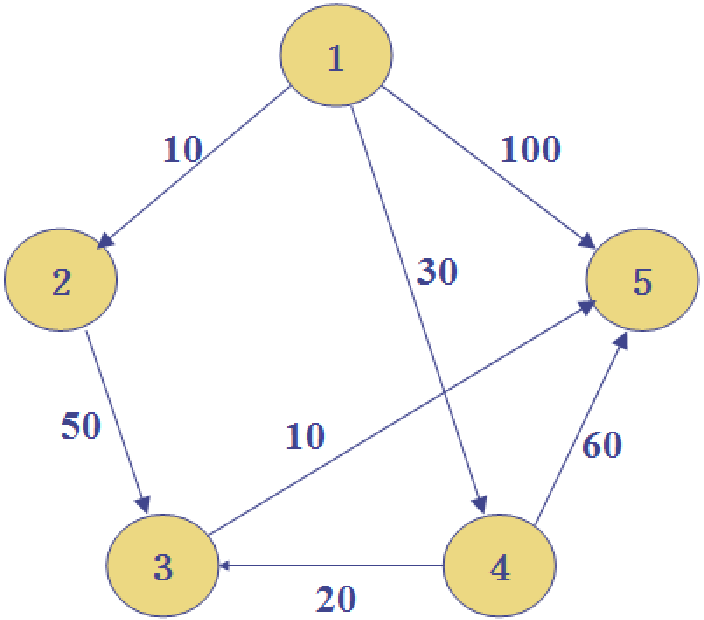
- 递归方法：$t(k)=3t(k-1)+c$

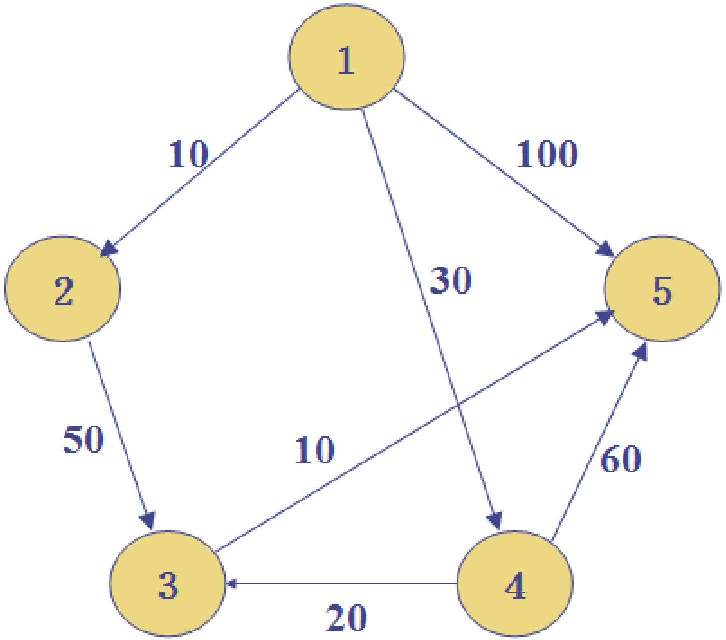
- 当注意到某些c（i, j, k-1）值可能被使用多次时，可以更高效地求解c（i, j, n）。利用避免重复计算c（i, j, k）的方法，可将计算c值的时间减少到$\Theta(n^3)$。


1
10
100
2
30
5
50
10
60
3
4
20

Floyd算法

$$A_0 = \begin{bmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$





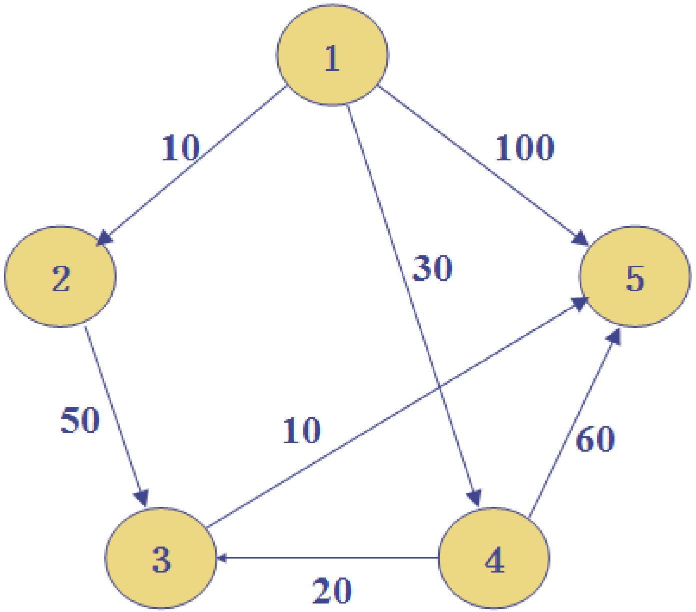
Floyd算法

$$A_0 = \begin{bmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

由A0经过顶点2得到A2：因为只有顶点1能到顶点2，顶点2只能到达顶点3，所以只有一个改变：

$$A_2 = \begin{bmatrix} 0 & 10 & 60 & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

C[1][3]=min{c[1][3],c[1][2]+c[2][3]}



Floyd算法

$$A_2 = \begin{bmatrix} 0 & 10 & 60 & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

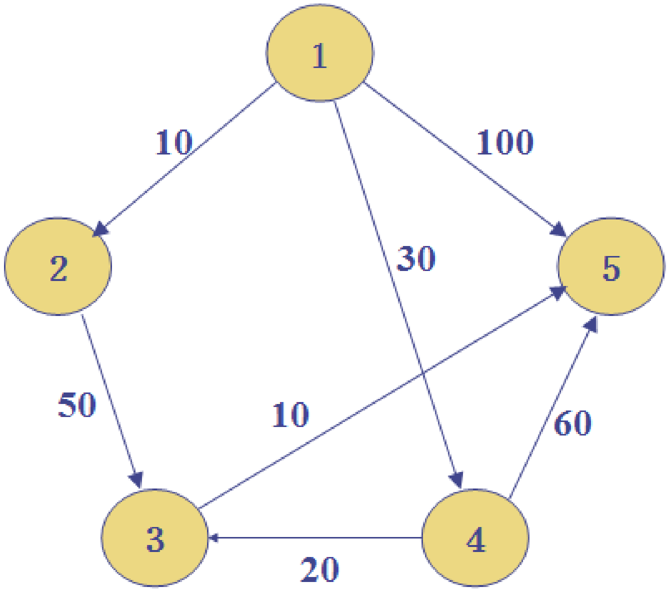
由A2经过顶点3得到A3：因为顶点1,2,4能到顶点3，顶点3只能到达顶点5，所以改变如下：

$$A_3 = \begin{bmatrix} 0 & 10 & 60 & 30 & 70 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

C[1][5]=c[1][3]+c[3][5]

C[2][5]=c[2][3]+c[3][5]

C[4][5]=c[4][3]+c[3][5]



Floyd算法

$$A_3 = \begin{bmatrix} 0 & 10 & 60 & 30 & 70 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

由A3经过顶点4得到A4：因为顶点1能到顶点4，顶点4能到达顶点3和5，所以改变如下：

C[1][5]=c[1][4]+c[4][5]

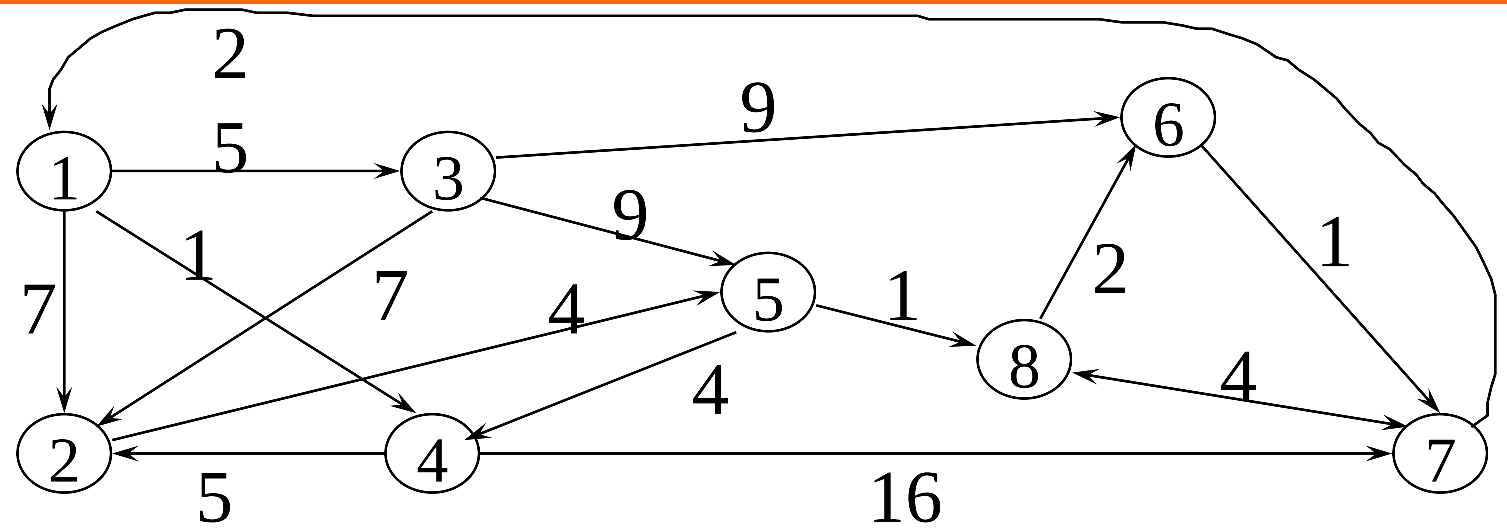
C[1][3]=c[1][4]+c[4][3]

$$A_4 = \begin{bmatrix} 0 & 10 & 50 & 30 & 60 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

$$\mathbf{A_0} = \begin{bmatrix} \mathbf{0} & \mathbf{10} & \infty & \mathbf{30} & \mathbf{100} \\ \infty & \mathbf{0} & \mathbf{50} & \infty & \infty \\ \infty & \infty & \mathbf{0} & \infty & \mathbf{10} \\ \infty & \infty & \mathbf{20} & \mathbf{0} & \mathbf{60} \\ \infty & \infty & \infty & \infty & \mathbf{0} \end{bmatrix} \qquad A_3 = \begin{bmatrix} 0 & 10 & 60 & 30 & 70 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

$$\mathbf{A_2} = \begin{bmatrix} \mathbf{0} & \mathbf{10} & \mathbf{60} & \mathbf{30} & \mathbf{100} \\ \infty & \mathbf{0} & \mathbf{50} & \infty & \infty \\ \infty & \infty & \mathbf{0} & \infty & \mathbf{10} \\ \infty & \infty & \mathbf{20} & \mathbf{0} & \mathbf{60} \\ \infty & \infty & \infty & \infty & \mathbf{0} \end{bmatrix} \qquad A_4 = \begin{bmatrix} 0 & 10 & 50 & 30 & 60 \\ \infty & 0 & 50 & \infty & 60 \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 30 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

示例



-	7	5	1	-	-	-	-
-	-	-	-	4	-	-	-
-	7	-	-	9	9	-	-
-	5	-	-	-	-	16	-
-	-	-	4	-	-	-	1
-	-	-	-	-	-	1	-
2	-	-	-	-	-	-	4
-	-	-	-	-	2	4	-

初始耗费矩阵

c(*,*) = c(*,*,0)

最终耗费矩阵 $c(*,*)=c(*,*,n)$

- 0 6 5 1 10 13 14 11
- 10 0 15 8 4 7 8 5
- 12 7 0 13 9 9 10 10
- 15 5 20 0 9 12 13 10
- 6 9 11 4 0 3 4 1
- 3 9 8 4 13 0 1 5
- 2 8 7 3 12 6 0 4
- 5 11 10 6 15 2 3 0
  - 1 到 7的最短路径长度14 .

kay 矩阵

- 0 4 0 0 4 8 8 5
- 8 0 8 5 0 8 8 5
- 7 0 0 5 0 0 6 5
- 8 0 8 0 2 8 8 5
- 8 4 8 0 0 8 8 0
- 7 7 7 7 7 0 0 7
- 0 4 1 1 4 8 0 0
- 7 7 7 7 7 0 6 0

确定最短路径

- 0 4 0 0 0 4 8 8 5
- 8 0 8 5 0 8 8 5
- 7 0 0 5 0 0 6 5
- 8 0 8 0 2 8 8 5
- 8 4 8 0 0 8 8 0
- 7 7 7 7 7 0 0 7
- 0 4 1 1 4 8 0 0
- 7 7 7 7 7 0 6 0

- 1 到 7的最短路径是：
- 1 4 2 5 8 6 7.

输出最短路径

```
template<class T>
void outputPath(T **c, int **kay, T noEdge, int  i, int  j)
{// 输出从i 到j的最短路径
    if (c[i][j] = = noEdge) {
        cout << "There is no path from " << i << " to " << j << endl;
        return;}
    cout << "The path is" << endl;
    cout << i << ' ';
    outputPath(kay, i, j);
    cout << endl;
}
```

输出最短路径

```
void outputPath(int **kay, int i, int j)
{//输出i 到j 的路径的实际代码
  // 不输出路径上的第一个顶点 (i)
if (i = = j) return;
if (kay[i][j] = = 0) //路径上没有中间顶点
    cout<<j << ' ';
else {// kay[i][j]是路径上的中间顶点
     outputPath(kay,  i,  kay[i][j]);
     outputPath(kay, kay[i][j], j);}
}
```

练习

- 以下图为例，
- （1）按Dijkstra算法计算从源点1到其它各个顶点的最短路径和最短路径长度。
- （2）使用Floyd算法计算各对顶点之间的最短路径。