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。

最优子结构性质：如果问题的最优解所包含的子问题的解也是最优的，我们就称该问题具有最优子结构性质（即满足最优化原理）。最优子结构性质为动态规划算法解决问题提供了重要线索。

0/1背包问题

- 在0/1背包问题中，需对容量为c 的背包进行装载。从n 个物品中选取装入背包的物品，每件物品i 的重量为w_i，价值为p_i。
- 可行的背包装载：背包中物品的总重量不能超过背包的容量
  - 约束条件为$\sum w_i x_i <= c$和$x_i \in [0,1] (1 \le i \le n)$。
- 最佳装载是指所装入的物品价值最高，即
- $\sum_{i=1}^{n} p_i x_i$ 取得最大值。

0/1背包问题

- 需求出x_i 的值
  - $x_i = 1$ 表示物品i 装入背包中
  - $x_i = 0$ 表示物品i 不装入背包

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$i = 1, 2, \cdots, n$ 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，$[x_2, \cdots, x_n]$ 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：

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

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

- 0/1背包问题：求解$f(1,c)$

- f(1,c)是初始时背包问题最优解的值。可通过公式(2)递归或者迭代来求解f(1,c)
- 从f(n,*)开始迭代， f(n,*)由公式（1）得出，然后由公式（2）递归计算f(i, *) （i=n-1, n-2, …, 2）,最后由公式(2)得出f(1, c)
- 例如，n=3, w=[100,14,10], p=[20,18,15], c=116

对于例 19-2，若 $0 \leqslant y<10$，则 $f(3,y)=0$；若 $y \geqslant 10$，$f(3,y)=15$。利用递归公式（19-2），可得 $f(2,y)=0(0 \leqslant y<10)$；$f(2,y)=15$（$10 \leqslant y<14$）；$f(2,y)=18$（$14 \leqslant y<24$）和 $f(2,y)=33$（$y \geqslant 24$）。因此最优解 $f(1,116)=\max\{f(2,116), f(2,116-w_1)+p_1\}=\max\{f(2,116), f(2,16)+20\}=\max\{33,38\}=38$。

现在计算 x_i 值，步骤如下：若 $f(1,c)=f(2,c)$，则 $x_1=0$，因为我们可以用容量 c 装载物品 $2,\cdots,n$，而得到 $f(1,c)$ 的值。如果 $f(1,c) \neq f(2,c)$，则 $x_1=1$。接下来需从剩余容量 $c-w_1$ 中寻求最优解，用 $f(2,c-w_1)$ 表示。依此类推，可得到所有的 $x_i(i=1,\cdots,n)$ 值。

在该例中，因为 $f(2,116)=33 \neq f(1,116)$，所以 $x_1=1$。接着利用返回值 $38-p_1=18$ 和最大容量 $116-w_1=16$ 来确定 x_2 及 x_3。注意，$f(2,16)=18$。因为 $f(3,16)=15 \neq f(2,16)$，所以 $x_2=1$，剩余容量为 $16-w_2=2$。因为 $f(3,2)=0$，所以 $x_3=0$。■

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

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

程序 19-1 背包问题的递归函数

时间$O(2^n)$

```
int f(int i,int theCapacity)
{// 返回 f(i, theCapacity) 的值
   if(i == numberOfObjects)
     return(theCapacity < weight[numberOfObjects]
           ? 0:profit[numberOfObjects]);
   if(theCapacity<weight[i])
     return f(i+1,theCapacity);
   return max(f(i+1,theCapacity),
              f(i+1,theCapacity-weight[i])+profit[i]);
}
```

应用动态规划求解的步骤如下：

1）证实最优原则是适用的。

2）建立动态规划的递归方程式。

3）求解动态规划的递归方程式以获得最优解。

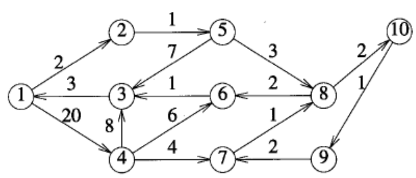
4）沿着最优解的生成过程进行**回溯**（traceback）。

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 =$n(n$-1) **条最短路径**。

使用Dijkstra算法

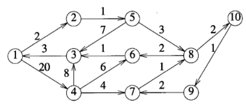
- Dijkstra算法：边上的权值>=0
- 使用Dijkstra算法 n 次, 每次用1个顶点作为源点
- 时间复杂性： $O(n^3)$.
- 回忆Dijkstra 算法



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**允许经过的顶点编号都不大于k**。
- $$c(i,j,0)=\begin{cases} \text{边 (i,j)的长度} & (i,j) \in E \\ 0 & i=j \\ +\infty \ \text{(noEdge)} & \text{其它} \end{cases}$$
- 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



- 若k=0,1,2,3, 则c(1,3,k)=∞；c(1,3,4)=28;
- 若k=5,6,7, 则c(1,3,k)=10；若k=8,9,10, 则c(1,3,k)=9
- c(i,j,0) ⇒ c(i,j,n) ？
- c(i,j,k-1)⇒ c(i,j,k), k > 0 ？

c(i, j, k-1)⇒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***c*(*i, j, k*-1)**
- 2. 该路径含中间顶点*k*，路径长度为
- ***c*(*i, k, k*-1) +*c* (*k, j, k*- 1)**。

- 结合以上两种情况，*c*(*i, j, k*) 取两者中的最小值
  ⇒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, …, n的顺序计算c(i,j,k)

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。kay值可以用来构建从一个顶点到另一个顶点的最短路径
- 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

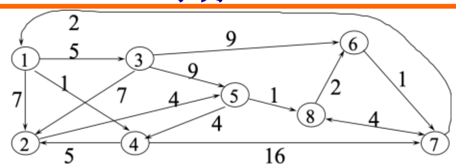
```
template<class T>
void  Allpairs(T **c, int **kay)
{//所有点对的最短路径;对于所有i和j，计算c[i][j]和kay[i][j]
//初始化c[i][j]=c(i，j，0)
for (int i=1;i<=n;i++)
         for (int j=1;j<=n;j++){
                  c[i][j]=a[i][j];
                  kay[i][j]=0;}
for  (i=1;i<=n;i++)
         c[i][i]=0;
```


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： $\Theta(n^3)$.

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 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);}
}
```

- 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

- 1 到 7的最短路径是：
- 1 4 2 5 8 6 7.

1-------------7

1--------8----7

1------5-8-6-7

1-4---5-8-6-7 kay[5][8]=0

kay[8][6]=0 kay[6][7]=0

1-4-2-5-8-6-7 kay[1][4]=0

1-4-2-5-8-6-7 kay[4][2]=0

kay[2][5]=0