《计算复杂性理论》
第9讲 动态规划方法(2)

山东师范大学信息科学与工程学院
段会川
2014年11月

目录

0-1背包问题—形式化定义

- 给定n个重量为$w_1, w_2, \cdots, w_n$价值为$v_1, v_2, \cdots, v_n$的物品和容量为W的背包，其中$W < \sum_{i=1}^{n} w_i$且物品不可分割，问怎样装入物品可以获得最大的价值？
- 以$x_1, x_2, \cdots, x_n$表示物品的装入情况，其中$x_i \in \{0, 1\}$，则0-1背包问题可以表达为如下所示的优化问题：

$$\max_{x_1, x_2, \cdots, x_n} V(x_1, x_2, \cdots, x_n) = \sum_{i=1}^{n} x_i v_i,$$

$$s.t. \quad \sum_{i=1}^{n} x_i w_i \le W,$$

$$x_i \in \{0, 1\}, i = 1, 2, \cdots, n.$$

0-1背包问题最优子结构性质分析

假设$(x_1, x_2, \cdots, x_n)$是所给0-1背包问题的一个最优解，则$(x_2, \cdots, x_n)$是下面相应子问题的一个最优解：

$$约束条件：\begin{cases} \sum_{i=2}^{n} w_i x_i \leqslant W - w_1 x_1 \\ x_i \in \{0, 1\} \quad 2 \leqslant i \leqslant n \end{cases}，\quad 目标函数：\max \sum_{i=2}^{n} v_i x_i。$$

证明：(反证法)设$(x_2, \cdots, x_n)$不是上述子问题的一个最优解，而$(y_2, \cdots, y_n)$是上述子问题的一个最优解，则最优解向量$(y_2, \cdots, y_n)$所求得的目标函数的值要比解向量$(x_2, \cdots, x_n)$求得的目标函数的值要大，即

$$\sum_{i=2}^{n} v_i y_i > \sum_{i=2}^{n} v_i x_i \qquad (4\text{-}9)$$

王秋芬 P100

0-1背包问题最优子结构性质分析

证明：(反证法)设$(x_2, \cdots, x_n)$不是上述子问题的一个最优解，而$(y_2, \cdots, y_n)$是上述子问题的一个最优解，则最优解向量$(y_2, \cdots, y_n)$所求得的目标函数的值要比解向量$(x_2, \cdots, x_n)$求得的目标函数的值要大，即

$$\sum_{i=2}^{n} v_i y_i > \sum_{i=2}^{n} v_i x_i \qquad (4\text{-}9)$$

又因为最优解向量$(y_2, \cdots, y_n)$满足约束条件：$\sum_{i=2}^{n} w_i y_i \leqslant W - w_1 x_1$，即$w_1 x_1 + \sum_{i=2}^{n} w_i y_i \leqslant W$，这说明$(x_1, y_2, \cdots, y_n)$是原问题的一个解。此时，在式(4-9)的两边同时加上$v_1 x_1$，可得不等式$v_1 x_1 + \sum_{i=2}^{n} v_i y_i > v_1 x_1 + \sum_{i=2}^{n} v_i x_i = \sum_{i=1}^{n} v_i x_i$，这说明在原问题的两个解$(x_1, y_2, y_3, \cdots, y_n)$和$(x_1, x_2, x_3, \cdots, x_n)$中，前者比后者所代表的装入背包的物品总价值要大，即$(x_1, x_2, x_3, \cdots, x_n)$不是原问题的最优解。这与$(x_1, x_2, x_3, \cdots, x_n)$是原问题的最优解矛盾。故$(x_2, \cdots, x_n)$是上述相应子问题的一个最优解，最优子结构性质得证。

王秋芬 P100

0-1背包问题最优值的递归关系式

由于0-1背包问题的解是用向量$(x_1, x_2, \cdots, x_n)$来描述的。因此，该问题可以看做是决策一个n元0-1向量$(x_1, x_2, \cdots, x_n)$。对于任意一个分量x_i的决策是“决定$x_i=1$或$x_i=0$”，$i=1,2,\cdots,n$。对x_{i-1}决策后，序列$(x_1, x_2, \cdots, x_{i-1})$已被确定，在决策$x_i$时，问题处于下列两种状态之一：

(1) 背包容量不足以装入物品i，则$x_i=0$，装入背包的价值不增加。

(2) 背包容量足以装入物品i，则$x_i=1$，装入背包的价值增加v_i。

在这两种情况下，装入背包的价值最大者应该是对x_i决策后的价值。

令$C[i][j]$表示子问题$\begin{cases} \sum_{k=1}^{i} w_k x_k \leqslant j \\ x_k \in \{0, 1\} \quad 1 \leqslant k \leqslant i \end{cases}$的最优值，即$C[i][j] = \max \sum_{k=1}^{i} v_k x_k$。那么，$C[i-1][j-w_i x_i]$表示该问题的子问题$\begin{cases} \sum_{k=1}^{i-1} w_k x_k \leqslant j - w_i x_i \\ x_k, x_i \in \{0, 1\} \quad 1 \leqslant k \leqslant i-1 \end{cases}$的最优值。

王秋芬 P100-1

0-1背包问题最优值的递归关系式

如果 $j=0$ 或 $i=0$，令 $C[0][j]=C[i][0]=0$，$1\leqslant i\leqslant n$，$1\leqslant j\leqslant W$；如果 $j<w_i$，第 i 个物品肯定不能装入背包，$x_i=0$，此时 $C[i][j]=C[i-1][j-w_ix_i]=C[i-1][j]$；如果 $j\geqslant w_i$，第 i 个物品能够装入背包：如果第 i 个物品不装入背包，即 $x_i=0$，则 $C[i][j]=C[i-1][j-w_ix_i]=C[i-1][j]$；如果第 i 个物品装入背包，即 $x_i=1$，则 $C[i][j]=C[i-1][j-w_ix_i]+v_i=C[i-1][j-w_i]+v_i$。可见当 $j\geqslant w_i$ 时，$C[i][j]$ 应取二者的最大值，即 $\max\{C[i-1][j],C[i-1][j-w_i]+v_i\}$。

由此可得最优值的递归定义式为：

$$C[0][j]=C[i][0]=0 \tag{4-10}$$

$$C[i][j]=\begin{cases}C[i-1][j] & j<w_i\\ \max\{C[i-1][j],C[i-1][j-w_i]+v_i\} & j\geqslant w_i\end{cases} \tag{4-11}$$

王秋芬 P101

0-1背包问题DP算法设计

求解 0-1 背包问题的算法步骤如下：

步骤 1：设计算法所需的数据结构。采用数组 $w[n]$ 来存放 n 个物品的重量；数组 $v[n]$ 来存放 n 个物品的价值，背包容量为 W，数组 $C[n+1][W+1]$ 来存放每一次迭代的执行结果；数组 $x[n]$ 用来存储所装入背包的物品状态。

步骤 2：初始化。按式(4-10)初始化数组 C。

步骤 3：循环阶段。按式(4-11)确定前 i 个物品能够装入背包的情况下得到的最优值。

步骤 3-1：$i=1$ 时，求出 $C[1][j]$，$1\leqslant j\leqslant W$。

步骤 3-2：$i=2$ 时，求出 $C[2][j]$，$1\leqslant j\leqslant W$。

以此类推，直到……

步骤 3-n：$i=n$ 时，求出 $C[n][W]$。此时，$C[n][W]$ 便是最优值。

王秋芬 P101

0-1背包问题DP算法设计

步骤 4：确定装入背包的具体物品。从 $C[n][W]$ 的值向前推，如果 $C[n][W]>C[n-1][W]$，表明第 n 个物品被装入背包，则 $x_n=1$，前 $n-1$ 个物品被装入容量为 $W-w_n$ 的背包中；否则，第 n 个物品没有被装入背包，则 $x_n=0$，前 $n-1$ 个物品被装入容量为 W 的背包中。以此类推，直到确定第 1 个物品是否被装入背包中为止。由此，得到以下关系式：

$$\begin{cases}x_i=0, \quad j=j & \text{当 } C[i][j]=C[i-1][j]\\ x_i=1, \quad j=j-w_i & \text{当 } C[i][j]>C[i-1][j]\end{cases} \tag{4-12}$$

按照式(4-12)，从 $C[n][W]$ 的值向前倒推，即 j 初始为 W，i 初始为 n，即可确定装入背包的具体物品。

王秋芬 P101

0-1背包问题DP算法设计

Bottom-up computation: Computing the table using

$$V[i,w]=\max(V[i-1,w],v_i+V[i-1,w-w_i])$$

row by row.

V[i,w]	w=0	1	2	3	…	…	W	
i= 0	0	0	0	0	…	…	0	bottom
1								
2								
⋮								
n								

up

Lecture 13: The Knapsack Problem, P12
http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

0-1背包问题DP求解实例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

W=10

王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

采用二维数组 $C[6][11]$ 来存放各个子问题的最优值，行 i 表示物品，列 j 表示背包容量，表中数据表示 $C[i][j]$。

(1) 根据式(4-10)初始化第 0 行和第 0 列，如表 4-12 所示。

表 4-12　初始化第 0 行和第 0 列

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										
5	0										

王秋芬 P102

0-1背包问题DP求解实例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

W=10

王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

(2) $i=1$ 时，求出 $C[1][j]$，$1\leqslant j\leqslant W$。

由于物品 1 的重量 $w_1=2$，价值 $v_1=6$，根据式(4-11)，分两种情况讨论。

① 如果 $j<w_1$，即 $j<2$ 时，$C[1][j]=C[0][j]$。

② 如果 $j\geqslant w_1$，即 $j\geqslant 2$ 时，$C[1][j]=\max\{C[0][j],C[0][j-w_1]+v_1\}=\max\{C[0][j],C[0][j-2]+6\}$。

$i=1$ 时的内容如表 4-13 所示。

表 4-13　$i=1$ 时的内容

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0										
3	0										
4	0										
5	0										

0-1背包问题DP求解实例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

W=10

王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

(3) $i=2$ 时，求出 $C[2][j]$，$1\leqslant j\leqslant W$。

由于物品 2 的重量 $w_2=2$，价值 $v_2=3$，根据式(4-11)，分两种情况讨论。

① 如果 $j<w_2$，即 $j<2$ 时，$C[2][j]=C[1][j]$。

② 如果 $j\geqslant w_2$，即 $j\geqslant 2$ 时，$C[2][j]=\max\{C[1][j], C[1][j-w_2]+v_2\}=\max\{C[1][j], C[1][j-2]+3\}$。

由于 j 的取值不同，满足的条件也就有所不同，$i=2$ 时的内容如表 4-14 所示。

表 4-14　$i=2$ 时的内容

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0										
4	0										
5	0										

0-1背包问题DP求解实例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

W=10

王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

(4) $i=3$ 时，求出 $C[3][j]$，$1\leqslant j\leqslant W$。

由于物品 3 的重量 $w_3=6$，$v_3=5$，根据式(4-11)，分两种情况讨论。

① 如果 $j<w_3$，即 $j<6$ 时，$C[3][j]=C[2][j]$。

② 如果 $j\geqslant w_3$，即 $j\geqslant 6$ 时，$C[3][j]=\max\{C[2][j], C[2][j-w_3]+v_3\}=\max\{C[2][j], C[2][j-6]+5\}$。

$i=3$ 时的内容如表 4-15 所示。

表 4-15　$i=3$ 时的内容

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0										
5	0										

0-1背包问题DP求解实例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

W=10

王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

(5) $i=4$ 时，求出 $C[4][j]$，$1\leqslant j\leqslant W$。

由于物品 4 的重量 $w_4=5$，$v_4=4$，根据式(4-11)，分两种情况讨论。

① 如果 $j<w_4$，即 $j<5$ 时，$C[4][j]=C[3][j]$。

② 如果 $j\geqslant w_4$，即 $j\geqslant 5$ 时，$C[4][j]=\max\{C[3][j], C[3][j-w_4]+v_4\}=\max\{C[3][j], C[3][j-5]+4\}$。

$i=4$ 时的内容如表 4-16 所示。

表 4-16　$i=4$ 时的内容

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14
5	0										

0-1背包问题DP求解实例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

W=10

王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

(6) $i=5$ 时，求出 $C[5][j]$，$1\leqslant j\leqslant W$，即进行 $i=5$ 行的填表。

由于物品 5 的重量 $w_5=4$，$v_5=6$，根据式(4-11)，分两种情况讨论。

① 如果 $j<w_5$，即 $j<4$ 时，$C[5][j]=C[4][j]$。

② 如果 $j\geqslant w_5$，即 $j\geqslant 4$ 时，$C[5][j]=\max\{C[4][j], C[4][j-w_5]+v_5\}=\max\{C[4][j], C[4][j-4]+6\}$。

由于 j 的取值不同，满足的条件也就有所不同，$i=5$ 时的内容如表 4-17 所示。

0-1背包问题DP求解实例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

W=10

王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

表 4-17　$i=5$ 时的内容

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14
5	0	0	6	6	9	9	12	12	15	15	15

最终，从表 4-17 中可以看出，装入背包的物品的最大价值是 15。

0-1背包问题DP求解实例

编号	1	2	3	4	5
重量	2	2	6	5	4
价值	6	3	5	4	6

W=10

王秋芬，P101

解：1,1,0,0,1
重量：2+2+4=8
价值：6+3+6=15

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14
5	0	0	6	6	9	9	12	12	15	15	15

(7) 从 $C[n][W]$ 的值根据式(4-12)向前推，最终可求出装入背包的具体物品，即问题的最优解。

初始时，$j=W$，$i=5$。

如果 $C[i][j]=C[i-1][j]$，说明第 i 个物品没有被装入背包，则 $x_i=0$。

如果 $C[i][j]>C[i-1][j]$，说明第 i 个物品被装入背包，则 $x_i=1$，$j=j-w_i$。

0-1背包问题DP求解实例

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	6	6	6	6	6	6	6	6	6
2	0	0	6	6	9	9	9	9	9	9	9
3	0	0	6	6	9	9	9	9	11	11	14
4	0	0	6	6	9	9	9	10	11	13	14
5	0	0	6	6	9	9	12	12	15	15	15

由于 $C[n][W]=C[5][10]=15>C[4][10]=14$，说明物品 5 被装入了背包，因此 $x_5=1$，且更新 $j=j-w[5]=10-4=6$。由于 $C[4][j]=C[4][6]=9=C[3][6]$，说明物品 4 没有被装入背包，因此 $x_4=0$；由于 $C[3][j]=C[3][6]=9=C[2][6]=9$，说明物品 3 没有被装入背包，因此 $x_3=0$。由于 $C[2][j]=C[2][6]=9>C[1][6]=6$，说明物品 2 被装入了背包，因此 $x_2=1$，且更新 $j=j-w[2]=6-2=4$。由于 $C[1][j]=C[1][4]=6>C[0][4]=0$，说明物品 1 被装入了背包，因此 $x_1=1$，且更新 $j=j-w[1]=4-2=2$。最终可求得装入背包的物品的最优解 $X=(x_1,x_2,\cdots,x_n)=(1,1,0,0,1)$。

王秋芬 P104

0-1背包问题DP求解实例

Example of the Bottom-up computation

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
$i=0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

Lecture 13: The Knapsack Problem, P13
http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

0-1背包问题DP求解算法

```
int KnapSack(int n,int w[],int v[])      //物品个数 n、物品的价值 v[n]和物品的重量 w[n]
  {
    int i,j,C[n][n],x[n];
    for(i=0;i<=n;i++)
          C[i][0]=0;                    //初始化第 0 列
    for(i=0;i<=W;i++)
          C[0][i]=0;                    //初始化第 0 行
    for(i=1;i<=n;i++)                   //计算 C[i][j]
       for(j=1;j<=W;j++)
            if(j<w[i])
                C[i][j]=C[i-1][j];
           else
              C[i][j]=max(C[i-1][j],C[i-1][j-w[i]]+v[i]);
    //构造最优解
    j=W;
    for(i=n;i>0;i--)
      if(C[i][j]>C[i-1][j])
         {
          x[i]=1;
          j-=w[i];
          }
        else
          x[i]=0;
     return C[n][W];
  }
```


王秋芬 P104-5

0-1背包问题DP算法分析

```
for(i=1;i<=n;i++)                 //计算 C[i][j]
   for(j=1;j<=W;j++)
        if(j<w[i])
             C[i][j]=C[i-1][j];
       else
          C[i][j]=max(C[i-1][j],C[i-1][j-w[i]]+v[i]);
```

在算法 KnapSack 中，第三个循环是两层嵌套的 for 循环，为此，可选定语句 if($j<w[i]$)作为基本语句，其运行时间为 $n\times W$，由此可见，算法 KnapSack 的时间复杂性为 $O(nW)$。

该算法有两个较为明显的缺点：一是算法要求所给物品的重量 $w_i(1\leqslant i\leqslant n)$是整数；二是当背包容量 W 很大时，算法需要的计算时间较多，例如，当 $W>2^n$ 时，算法需要 $O(n2^n)$的计算时间。因此，在这里设计了对算法 KnapSack 的改进方法，采用该方法可克服这两大缺点。

王秋芬 P104-5

0-1背包问题DP算法伪代码及复杂度

```
KnapSack(v, w, n, W)
{
   for (w = 0 to W) V[0, w] = 0;
   for (i = 1 to n)
      for (w = 0 to W)
         if (w[i] ≤ w)
            V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]};
         else
            V[i, w] = V[i − 1, w];
   return V[n, W];
}
```

Time complexity: Clearly, $O(nW)$.

Lecture 13: The Knapsack Problem, P14
http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

0-1背包问题DP算法伪代码及复杂度

```
KnapSack(v, w, n, W)
{
   for (w = 0 to W) V[0, w] = 0;
   for (i = 1 to n)
      for (w = 0 to W)
         if ((w[i] ≤ w) and (v[i] + V[i − 1, w − w[i]] > V[i − 1, w]))
         {
            V[i, w] = v[i] + V[i − 1, w − w[i]];
            keep[i, w] = 1;
         }
         else
         {
            V[i, w] = V[i − 1, w];
            keep[i, w] = 0;
         }
   K = W;
   for (i = n downto 1)
      if (keep[i, K] == 1)
      {
         output i;
         K = K − w[i];
      }
   return V[n, W];
}
```

Lecture 13: The Knapsack Problem, P14
http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

0-1背包问题DP算法复杂度

- A polynomial-time algorithm is one that runs in time polynomial *in the total number of bits required to write out the input to the problem*.
- How many bits are required to write out the value W?
  - Answer: **O(log W)**.
- Therefore, $O(nW)$ is **exponential** in the number of bits required to write out the input.
  - Example: Adding one more bit to the end of the representation of W doubles its size and doubles the runtime.
- This algorithm is called a ***pseudopolynomial time algorithm***, since it is a polynomial in the *numeric value* of the input, not the number of bits in the input.

Intractable Problems, Part Two, P13
http://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/20/Small20.pdf

0-1背包问题DP算法复杂度

- The runtime of $O(nW)$ is better than our old runtime of $O(2^n n)$ assuming that $W = o(2^n)$.
  - That's *little-o*, not big-O.
- In fact – for *any* fixed W, this algorithm runs in linear time!
- Although there are exponentially many subsets to test, we can get away with just linear work if W is fixed!

Intractable Problems, Part Two, P14
http://web.stanford.edu/class/archive/cs/cs161/cs161.1138/lectures/20/Small20.pdf

目录

TSP问题的DP算法

例 6.1 货郎担问题。

如果对任意数目的 n 个城市，分别用 $1\sim n$ 的数字编号，则这个问题归结为在有向赋权图 $G=<V,E>$ 中，寻找一条路径最短的哈密尔顿回路问题。其中，$V=\{1,2,\cdots,n\}$ 表示城市顶点；边 $(i,j)\in E$ 表示城市 i 到城市 j 的距离，$i,j=1,2,\cdots,n$。这样，可以用图的邻接矩阵 C 来表示各个城市之间的距离，把这个矩阵称为费用矩阵。如果 $(i,j)\in E$，则 $c_{ij}>0$；否则，$c_{ij}=\infty$。

TSP问题的DP算法

令 $d(i,\overline{V})$ 表示从顶点 i 出发，经 $\overline{V}$ 中各个顶点一次，最终回到初始出发点的最短路径的长度。开始时，$\overline{V}=V-\{i\}$。于是，可以定义下面的动态规划函数：

$$d(i,V-\{i\})=d(i,\overline{V})=\min_{k\in\overline{V}}\{c_{ik}+d(k,\overline{V}-\{k\})\} \qquad (6.1.1)$$

$$d(k,\varphi)=c_{ki} \qquad k\neq i \qquad (6.1.2)$$

下面用 4 个城市的例子，来说明动态规划方法解货郎担问题的工作过程。假定 4 个城市的费用矩阵是：

$$C=(c_{ij})=\begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \infty \end{pmatrix}$$

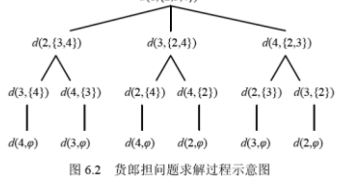


图 6.2 货郎担问题求解过程示意图

TSP问题的DP算法

$$C=(c_{ij})=\begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \infty \end{pmatrix}$$

根据式（6.1.1），由城市 1 出发，经城市 2、3、4，然后返回 1 的最短路径长度为：

$$d(1,\{2,3,4\})=\min\{c_{12}+d(2,\{3,4\}),c_{13}+d(3,\{2,4\}),c_{14}+d(4,\{2,3\})\}$$

这是最后一个阶段的决策，它必须依据 $d(2,\{3,4\}),d(3,\{2,4\}),d(4,\{2,3\})$ 的计算结果。于是，有：

$$d(2,\{3,4\})=\min\{c_{23}+d(3,\{4\}),c_{24}+d(4,\{3\})\}$$
$$d(3,\{2,4\})=\min\{c_{32}+d(2,\{4\}),c_{34}+d(4,\{2\})\}$$
$$d(4,\{2,3\})=\min\{c_{42}+d(2,\{3\}),c_{43}+d(3,\{2\})\}$$

这一阶段的决策，又必须依据下面的计算结果：

$$d(3,\{4\}),d(4,\{3\}),d(2,\{4\}),d(4,\{2\}),d(2,\{3\}),d(3,\{2\})$$

TSP问题的DP算法

再向前倒推，有：

$$\begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \infty \end{pmatrix}$$

$$d(3,\{4\}) = c_{34} + d(4,\varphi) = c_{34} + c_{41} = 2+3 = 5$$
$$d(4,\{3\}) = c_{43} + d(3,\varphi) = c_{43} + c_{31} = 5+6 = 11$$
$$d(2,\{4\}) = c_{24} + d(4,\varphi) = c_{24} + c_{41} = 3+3 = 6$$
$$d(4,\{2\}) = c_{42} + d(2,\varphi) = c_{42} + c_{21} = 7+5 = 12$$
$$d(2,\{3\}) = c_{23} + d(3,\varphi) = c_{23} + c_{31} = 2+6 = 8$$
$$d(3,\{2\}) = c_{32} + d(2,\varphi) = c_{32} + c_{21} = 4+5 = 9$$

有了这些结果，再向后计算，有：

$d(2,\{3,4\}) = \min\{2+5, 3+11\} = 7$ 路径顺序是：2,3,4,1

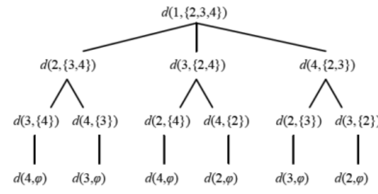
$d(3,\{2,4\}) = \min\{4+6, 2+12\} = 10$ 路径顺序是：3,2,4,1

$d(4,\{2,3\}) = \min\{7+8, 5+9\} = 14$ 路径顺序是：4,3,2,1

最后：

$d(1,\{2,3,4\}) = \min\{3+7, 6+10, 7+14\} = 10$ 路径顺序是：1,2,3,4,1

TSP问题的DP算法



$$\begin{pmatrix} \infty & 3 & 6 & 7 \\ 5 & \infty & 2 & 3 \\ 6 & 4 & \infty & 2 \\ 3 & 7 & 5 & \infty \end{pmatrix}$$

图 6.2 货郎担问题求解过程示意图

TSP问题的DP算法

$$d(i, V-\{i\}) = d(i,\overline{V}) = \min_{k\in\overline{V}}\{c_{ik} + d(k,\overline{V}-\{k\})\} \qquad (6.1.1)$$

$$d(k,\varphi) = c_{ki} \qquad k \neq i \qquad (6.1.2)$$

令 N_i 是计算式(6.1.1)时(从顶点 i 出发，返回顶点 i)所需要计算的形式为 $d(k,\overline{V}-\{k\})$ 的个数。开始计算 $d(i,V-\{i\})$ 时，集合 $V-\{i\}$ 中有 $n-1$ 个城市。以后，在计算 $d(k,\overline{V}-\{k\})$ 时，集合 $\overline{V}-\{k\}$ 的城市数目，在不同的决策阶段分别为 $n-2,\cdots,0$。在整个计算中，需要计算大小为 j 的不同城市集合的个数为 C_{n-1}^j， $j=0,1,\cdots,n-1$。因此，总个数为：

$$N_i = \sum_{j=0}^{n-1} C_{n-1}^j$$

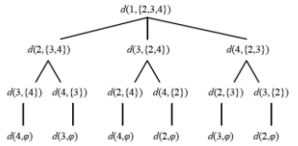


图 6.2 货郎担问题求解过程示意图

TSP问题的DP算法

$$N_i = \sum_{j=0}^{n-1} C_{n-1}^j$$

当 $\overline{V}-\{k\}$ 集合中的城市个数为 j 时，为了计算 $d(k,\overline{V}-\{k\})$，需要进行 j 次加法运算和 $j-1$ 次比较运算。因此，从 i 城市出发，经其他城市再回到 i，总的运算时间 T_i 为：

$$T_i = \sum_{j=0}^{n-1} j\cdot C_{n-1}^j < \sum_{j=0}^{n-1} n\cdot C_{n-1}^j = n\sum_{j=0}^{n-1} C_{n-1}^j$$

由二项式定理：

$$(x+y)^n = \sum_{j=0}^{n} C_n^j x^j y^{n-j}$$

令 $x=y=1$，可得：

$$T_i < n\cdot 2^{n-1} = O(n2^n)$$

则用动态规划方法求解货郎担问题，总的花费 T 为：

$$T = \sum_{i=1}^{n} T_i < n^2\cdot 2^{n-1} = O(n^2 2^n)$$

Bellman-Held-Karp algorithm

- The Held-Karp algorithm, also called Bellman-Held-Karp algorithm, is a dynamic programming algorithm proposed in 1962 independently by Bellman and by Held and Karp to solve the Traveling Salesman Problem (TSP).
- There is an optimization property for TSP:
  - Every subpath of a path of minimum distance is itself of minimum distance.

http://ucilnica1213.fmf.uni-lj.si/pluginfile.php/11706/mod_resource/content/0/HELDKarpAlgoritemZaPTP_clanek.pdf

Bellman-Held-Karp algorithm

- The Held-Karp algorithm, also called Bellman-Held-Karp algorithm, is a dynamic programming algorithm proposed in 1962 independently by Bellman and by Held and Karp to solve the Traveling Salesman Problem (TSP).
- There is an optimization property for TSP:
  - Every subpath of a path of minimum distance is itself of minimum distance.

Bellman-Held-Karp algorithm

Recursive formulation [edit]

Number the cities 1, 2, . . . , N and assume we start at city 1, and the distance between city i and city j is d_{ij} .Consider subsets $S \subseteq \{2, . . . , N\}$ of cities and, for $c \in S$, let D(S, c) be the minimum distance, starting at city 1, visiting all cities in S and finishing at city c.

First phase: if S = {c}, then $D(S, c) = d_{1,c}$. Otherwise: $D(S, c) = \min_{x\in S-c} (D(S - c, x) + d_{x,c})$

Second phase: the minimum distance for a complete tour of all cities is $M = \min_{c\in\{2,...,N\}} (D(\{2, . . . , N\}, c) + d_{c,1})$

A tour $n_1, . . . n_N$ is of minimum distance just when it satisfies $M = D(\{2, . . . , N\}, n_N) + d_{n_N,1}$.

Bellman-Held-Karp algorithm

Distance matrix: $C = \begin{pmatrix} 0 & 2 & 9 & 10 \\ 1 & 0 & 6 & 4 \\ 15 & 7 & 0 & 8 \\ 6 & 3 & 12 & 0 \end{pmatrix}$

$g(2, \varnothing) = c_{21} = 1$

$g(3, \varnothing) = c_{31} = 15$

$g(4, \varnothing) = c_{41} = 6$

k = 1, consider sets of 1 element: Set {2}:

$g(3,\{2\}) = c_{32} + g(2, \varnothing) = c_{32} + c_{21} = 7 + 1 = 8$ $\quad p(3,\{2\}) = 2$

$g(4,\{2\}) = c_{42} + g(2, \varnothing) = c_{42} + c_{21} = 3 + 1 = 4$ $\quad p(4,\{2\}) = 2$

Bellman-Held-Karp algorithm

Distance matrix: $C = \begin{pmatrix} 0 & 2 & 9 & 10 \\ 1 & 0 & 6 & 4 \\ 15 & 7 & 0 & 8 \\ 6 & 3 & 12 & 0 \end{pmatrix}$

$g(2, \varnothing) = c_{21} = 1$

$g(3, \varnothing) = c_{31} = 15$

$g(4, \varnothing) = c_{41} = 6$

Set {3}:

$g(2,\{3\}) = c_{23} + g(3, \varnothing) = c_{23} + c_{31} = 6 + 15 = 21$ $\quad p(2,\{3\}) = 3$

$g(4,\{3\}) = c_{43} + g(3, \varnothing) = c_{43} + c_{31} = 12 + 15 = 27$ $\quad p(4,\{3\}) = 3$

Set {4}:

$g(2,\{4\}) = c_{24} + g(4, \varnothing) = c_{24} + c_{41} = 4 + 6 = 10$ $\quad p(2,\{4\}) = 4$

$g(3,\{4\}) = c_{34} + g(4, \varnothing) = c_{34} + c_{41} = 8 + 6 = 14$ $\quad p(3,\{4\}) = 4$

Bellman-Held-Karp algorithm

- k = 2, consider sets of 2 elements: Set {2,3}:
  $g(4,\{2,3\}) = \min \{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\}$
  $= \min \{3+21, 12+8\} = \min \{24, 20\} = 20$
  $p(4,\{2,3\}) = 3$
- Set {2,4}:
  $g(3,\{2,4\}) = \min \{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\}$
  $= \min \{7+10, 8+4\} = \min \{17, 12\} = 12$
  $p(3,\{2,4\}) = 4$
- Set {3,4}:
  $g(2,\{3,4\}) = \min \{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\}$
  $= \min \{6+14, 4+27\} = \min \{20, 31\} = 20$
  $p(2,\{3,4\}) = 3$

Bellman-Held-Karp algorithm

- Length of an optimal tour:
  f = g(1,{2,3,4})
  = min { c12 + g(2,{3,4}), c13 + g(3,{2,4}), c14 + g(4,{2,3}) }
  = min {2 + 20, 9 + 12, 10 + 20}
  = min {22, 21, 30} = 21
- Successor of node 1: p(1,{2,3,4}) = 3
- Successor of node 3: p(3, {2,4}) = 4
- Successor of node 4: p(4, {2}) = 2
- Optimal TSP tour: 1 → 3 → 4 → 2 → 1

Bellman-Held-Karp algorithm

```
function algorithm T SP (G, n)
            for k := 2 to n do
               C({1, k}, k) := d1,k
            end for
            for s = 3 to n do
               for all S ⊆ {1, 2, . . . , n}, |S| = s do
                   for all k ∈ S do
                      {C(S, k) = minm≠1,m≠k,m∈S [C(S − {k}, m) + dm,k ]}
                   end for
               end for
            end for
            opt := mink≠1 [C({1, 2, 3, . . . , n}, k) + d1,k]
            return (opt)
  end
```

Bellman-Held-Karp algorithm

Faster than the exhaustive enumeration but still exponential, and the drawback of this algorithm, though, is that it also uses a lot of space: the worst-case time complexity of this algorithm is $O(2^n n^2)$ and the space $O(2^n n)$.

Time: the fundamental operations employed in the computation are additions and comparisons. The number of each in the first phase is given by

$$\left(\sum_{k=2}^{n-1} k(k-1)\binom{n-1}{k}\right) + (n-1) - (n-1)(n-2)2^{n-3} + (n-1)$$

and the number of occurrence of each in the second phase is $\sum_{k=2}^{n-1} k = \frac{n(n-1)}{2} - 1$

Space: $\left(\sum_{k=2}^{n-1} k\binom{n-1}{k}\right) + (n-1) = (n-1)2^{n-2}$

目录

- 0-1背包问题的DP算法
- TSP问题的DP算法

The End

Thanks!