算法设计与分析作业六

作者：吴润泽　　**学号：**181860109

Email：181860109@smail.nju.edu.cn

2020 年 5 月 18 日

目录

Chapter 13

problem 13.1

1)

```
FloydNext(G):
    D(0):=W /*初始情况下，两点间的最短路径长度就是它们的边权*/
    for(i:=1 to n):
        for(j:=1 to n):
            if map[i][j]!=Inf: /*如果ij间有边连接，则i的后继即为j
*/
                GO[i][j]=j
    for(k:=1 to n):
        for(i:=1 to n):
            for(j:=1 to n):
                if D[i][j]>D[i][k]+D[k][j]:
                    GO[i][j]=k /*如果i到j通过k的路径更短，则更新路
由表*/
                    D[i][j]=D[i][k]+D[k][j]
```

FloydNext.func

2) 算法更新路径 $i \to k \to j$，所选择的 j 的前驱与从选择的节点 k 到节点 j 的一条中间节点取自集合 $\{1, 2, \cdots, k-1\}$ 的最短路径上的前驱是一样的。

```
FloydPrev(G):
    D(0):=W
    for(i:=1 to n):
        for(j:=1 to n):
            if map[i][j]!=Inf: /*如果ij间有边连接，则j的前驱即为i
*/
                GO[i][j]:=i
    for(k:=1 to n):
        for(i:=1 to n):
            for(j:=1 to n):
                if D[i][j]>D[i][k]+D[k][j]:
                    GO[i][j]:=GO[k][j] /*如果i到j通过k的路径更短，
则前缀为Back[k][j]*/
                    D[i][j]:=D[i][k]+D[k][j]
```

FloydPrev.func

problem 13.2

1) 仿照 Dijkstra 算法，记录每条路径的最小值即可。

```
DijkstraCap(G):
    initialize the priority queue Fringe as empty
    insert some node v to Fringe
    cap:=[] /*cap 原来存放s到达其他点的吞吐量*/
    while Fringe!=empty:
        v:=Fringe.EXTRACT-MIN()
        cap[v]:=v.priority /*存放到达v的结果*/
        UPDATE-FRINGE(v,Fringe)
    return cap
UPDATE-FRINGE(v,Fringe):
    for neighbor w of v:
        w.priority:=min(vw.weight,v.priority)
        /*取路径的最小值，即当前路径已知吞吐量*/
        if w is UNSEEN: /*如果是第一次遍历到*/
            Fringe.INSERT(w,w.priority)  /*将w加入队列*/
        else:
            Fringe.decrease(w,w.priority) /*将w的权值更新为w.
    priority*/
```

DijkstraCap.func

2) 仿照 Floyd 算法，对路径 $i \to k \to j$ 从 ij，ik，kj 中选取最小权边即可。

```
FloydCap(G):
D(0):=W /*开始的吞吐量记为相连边的边权值*/
for(k:=1 to n):
    for(i:=1 to n):
        for(j:=1 to n):
            D[i][j]:=min(D[i][j],D[i][k],D[k][j]) /*选取3者的最小
    值，作为最大吞吐量*/
```

FloydCap.func

problem 13.7

计算出所有点到达 v_0 的最短路径和 v_0 到达其他顶点的最短路径。则 uv 间的最短路径即为 u 到 v_0 的最短路径加上 v 到 v_0 的最短路径。

```
ShortByV(G,v):
    D:=W /*初始化所有最短路径为起始结点权值*/
    for(k:=1 to n): /*计算所有节点到达v的最短路径*/
        for(i:=1 to n):
            D[i][v]:=min(D[i][v],D[i][k]+D[k][v])
    for(k:=1 to n): /*计算v到达所有节点的最短路径*/
        for(i:=1 to n):
            D[v][i]:=min(D[v][i],D[v][k]+D[k][i])
    for(i:=1 to n):
        for(j:=1 to n):
            D[i][j]:=D[i][v]+D[v][j] /*最短路径为i到v加v到j的最短
    路径*/
```

ShortByV.func

Chapter 14

problem 14.2

解 取子问题为 $dp[i][j]$，含义为使用前 i 个数能否满足元素和为 j。对于 $dp[i][j]$ 要么使用了第 i 个元素，要么使用了前 i-1 个元素。故其递推式为:

$$dp[i][j] = (dp[i-1][j])||(dp[i][j-A[i]]).$$

基于此得到整数子集和算法，**时间复杂度** $O(nS)$，**空间复杂度** $O(nS)$:

```
SubSeqSum(A,S):
    dp:=[n+1][S+1] /*dp数组大小为(n+1)(S+1)的二维数组*/
    for i:=1 to n do: /*初始化布尔dp数组*/
        dp[i][0]:=dp[i][A[i]]:=true
    for i:=2 to n do:
        for j:=1 to S do:
            if A[i]>j: /*如果A[i]已经大于j，必不能包含A[i]*/
                dp[i][j]:=dp[i-1][j]
            else:
                dp[i][j]:=(dp[i-1][j])||(dp[i][j-A[i]])
    return dp[n][s] /*返回使用n个数能否达到s即可*/
```

SubSeqSum.func

problem 14.3

解 取子问题 $dp[i]$ 为正整数 i 变为 1 需要的最少操作数。根据上述规则，可根据子问题 i 与 2，3 的倍数关系，分成三种情况列出递推式：

$$\begin{cases} i = 3k\&\&i = 2k, & dp[i] = min(d[i-1], d[i/2], d[i/3]) + 1, \\ i = 2k\&\&i! = 3k, & dp[i] = min(d[i-1], d[i/2]) + 1, \\ i = 3k\&\&i! = 2k, & dp[i] = min(d[i-1], d[i/3]) + 1, \\ i! = 3k\&\&i! = 2k, & dp[i] = d[i-1] + 1. \end{cases}$$

基于此得到算法伪代码，**时间复杂度** $O(n)$ ，**空间复杂度** $O(n)$::

```
MiniOp(n):
    dp:=[n+1] /*dp数组大小为n+1的一维数组*/
    dp[1]:=dp[2]:=dp[3]:=1 /*1,2,3均只需要1次操作*/
    for i:=4 to n do:
        if i%6=0:
            dp[i]:=min(d[i-1],d[i/2],d[i/3])+1
        else if i%3=0:
            dp[i]:=min(d[i-1],d[i/3])+1
        else if i%2=0:
            dp[i]:=min(d[i-1],d[i/2])+1
        else:
            dp[i]:=d[i-1]+1
    return dp[n] /*返回子问题规模为n的最少操作数即可*/
```

MiniOp.func

problem 14.6

1) 取子问题 $dp[i][j]$ 记为 X 的前 i 个字符与 Y 的前 j 个字符的 LCS 长度。基于 $X[i]$ 与 $Y[j]$ 是否相等得到递推方程:

$$\begin{cases} X[i] == Y[j], & dp[i][j] = dp[i-1][j-1] + 1, \\ X[i]! = Y[j], & dp[i][j] = max(dp[i-1][j], dp[i][j-1]). \end{cases}$$

基于此得到算法伪代码，**时间复杂度** $O(mn)$ ，**空间复杂度** $O(mn)$:

```
LCS(X,Y):
    dp:=[m+1][n+1]  /*dp数组大小为(m+1)(n+1)的二维数组*/
    for i:=1 to m do:
        dp[i][0]:=0  /*X子串与空字符串的LCS为0*/
    for j:=1 to n do:
        dp[0][j]:=0  /*Y子串与空字符串的LCS为0*/
    for i:=1 to m do:
        for j:=1 to n do:
            if X[i]=Y[j]:
                dp[i][j]:=dp[i-1][j-1]+1
            else:
                dp[i][j]:=max(dp[i-1][j],dp[i][j-1])
    return dp[m][n]  /*返回子问题规模为X和Y的LCS即可*/
```

LCS.func

2) X 中的字符可以重复出现，当 $X[i] = Y[j]$ 时，对于子问题 $dp[i][j]$，其等价于 X 的前 i 个字符与 Y 前 j-1 个字符的 LCS 加 1。递推方程修改为:

$$\begin{cases} X[i] == Y[j], & dp[i][j] = dp[i][j-1] + 1, \\ X[i]! = Y[j], & dp[i][j] = max(dp[i-1][j], dp[i][j-1]). \end{cases}$$

```
ReLCS(X,Y):
    dp:=[m+1][n+1]  /*dp数组大小为(m+1)(n+1)的二维数组*/
    for i:=1 to m do:
        dp[i][0]:=0  /*X子串与空字符串的LCS为0*/
    for j:=1 to n do:
        dp[0][j]:=0  /*Y子串与空字符串的LCS为0*/
    for i:=1 to m do:
        for j:=1 to n do:
            if X[i]=Y[j]:
```

```
                dp[i][j]:=dp[i][j-1]+1 /*递推式进行修改*/
            else:
                dp[i][j]:=max(dp[i-1][j],dp[i][j-1])
    return dp[m][n] /*返回子问题规模为X和Y的LCS即可*/
```

ReLCS.func

时间复杂度与空间复杂度不变。

3) 在循环中记录重复次数，当 $X[i] = Y[j]$ 且 $X[i]$ 重复次数超过 k 后，$dp[i][j]$ 的 LCS 等价于 X 的前 i-1 个字符和 Y 前 j-1 个字符的 LCS 加 1。

```
LimReLCS(X,Y,k):
    dp:=[m+1][n+1] /*dp数组大小为(m+1)(n+1)的二维数组*/
    for i:=1 to m do:
        dp[i][0]:=0 /*X子串与空字符串的LCS为0*/
    for j:=1 to n do:
        dp[0][j]:=0 /*Y子串与空字符串的LCS为0*/
    repnum:=0 /*重复次数初始化为0*/
    for i:=1 to m do:
        for j:=1 to n do:
            if X[i]=Y[j]:
                if repnum<k:
                    repnum:=repnum+1
                    dp[i][j]:=dp[i][j-1]+1 /*递推式进行修改*/
                else:
                    dp[i][j]:=dp[i-1][j-1]+1
            else:
                repnum:=0 /*X[i]!=Y[j]即不再使用X[i]*/
                dp[i][j]:=max(dp[i-1][j],dp[i][j-1])
    return dp[m][n] /*返回子问题规模为X和Y的LCS即可*/
```

LimReLCS.func

problem 14.7

解 取子问题 $dp[i][j]$ 记为以 $T[i]$ 为结尾的前向串，以 $T[j]$ 为开始的后向串的最大连续子串长度。为了满足两子串不重叠，易得 $i<j$。根据 $T[i]$ 与 $T[j]$ 得递推方程:

$$\begin{cases} T[i] == T[j] \&\& i < j, & dp[i][j] = dp[i-1][j+1]+1, \\ T[i]! = T[j] \&\& i < j, & dp[i][j] = 0. \end{cases}$$

基于此得到算法伪代码，**时间复杂度** $O(n^2)$ ，**空间复杂度** $O(n^2)$:

```
FroLas(T):
    dp:=[n+1][n+1] /*dp数组大小为(m+1)(n+1)的二维数组*/
    res:=-Inf /*初始化为最小值，结果保存最大的连续长度*/
    for i:=1 to n do:
        dp[i][0]:=dp[0][i]:=0 /*子串与空字符串的重叠长度为0*/
    for i:=1 to n do:   /*前向串从头向后扩充*/
        for j:=n to i+1 do: /*前向串从尾部向前扩充*/
            if T[i]=T[j]:
                dp[i][j]:=dp[i-1][j+1]+1
            else:
                dp[i][j]:=0
            res:=max(res,dp[i][j]) /*更新结果*/
    return res
```

FroLas.func

problem 14.11

1) 与 problem 14.7类似，取子问题 $dp[i][j]$ 为子串 $T[i\cdots j]$ 的最长回文子序列。同样根据 $T[i]$ 与 $T[j]$ 得到递推方程:

$$\begin{cases} T[i] == T[j] \&\& i < j, & dp[i][j] = dp[i+1][j-1]+2, \\ T[i]! = T[j] \&\& i < j, & dp[i][j] = max(dp[i][j-1], dp[i+1][j]). \end{cases}$$

基于此得到算法伪代码，**时间复杂度** $O(n^2)$ ，**空间复杂度** $O(n^2)$:

```
PalindromSeq(T):
    dp:=[n+1][n+1] /*dp数组大小为(n+1)(n+1)的二维数组*/
    for i:=1 to n do:
        dp[i][i]:=1 /*单个字符的回文串长度为1*/
    for i:=1 to n do:
```

```
        for j:=i to n-i-1 do:  /*枚举i-j的子串*/
            if T[i]=T[j]:
                dp[i][j]:=dp[i+1][j-1]+2
            else:
                dp[i][j]:=max(dp[i][j-1],dp[i+1][j])
    return dp[0][n-1] /*返回原字符串的最大回文序列长度*/
```

PalindromSeq.func

2) 采用类似的算法,将字符串中的所有回文字串划分。取子问题 $IsPalind[i][j]$ 子串从 i 到 j 是否为回文子串，递推方程为:

$$IsPalind[i][j] = (IsPalind[i+1][j-1])\&(T[i] == T[j]).$$

取子问题 $dp[i]$ 为划分前 i 个字符串需要的划分次数，如果 $j-i$ 为回文子串，那么 $dp[i] \geq dp[j]+1$。因此递推方程为:

$$dp[i] = \min(dp[i], (IsPalind[j][i])\&\&(dp[j]+1)), j = 1, 2, \cdots, i-1.$$

基于此得到算法伪代码，**时间复杂度** $O(n^2)$ ，**空间复杂度** $O(n^2)$:

```
FindPalind(T):
    IsPalind[n+1][n+1]  /*IsPalind数组大小为(n+1)(n+1)的二维数组*/
    for i:=1 to n do:
        IsPalind[i][i]:=true  /*单个字符均为回文串*/
    for i:=1 to n do:
        for j:=1 to i do:  /*枚举j-i的子串*/
            IsPalind[i][j]:=(IsPalind[i+1][j-1])&(T[i]=T[j])

PalindSplit(T):
    FindPalind(T)  /*将字符串中所有的回文子串进行标记*/
    dp:=[n+1]  /*dp数组大小为(n+1)的一维数组*/
    for i:=1 to n do:
        dp[i]:=i-1  /*初始化划分i个字符需要i-1次*/
    for i:=1 to n do:
        for j:=1 to i do:   /*枚举j-i的子串*/
            if IsPalind[i][j]:
                dp[i]:=min(dp[i],dp[j]+1)
    return dp[n-1]+1 /*返回原字符串最小划分次数+1，即回文串数量*/
```

PalindSplit.func

problem 14.13

1) 取子问题 $dp[i]$ 记录是否可以表示金额 i。枚举使用每个硬币且不超过金额 v 的情况，如果 $dp[j]$ 为真，则 $dp[x_k]$ 也为真，递推方程为:

$$dp[j + x_k] := dp[j], k = 1, 2 \cdots, n.$$

```
RepCoin(X,v):
    dp:=[v+1] /*dp数组大小为(v+1)的一维数组*/
    for k:=1 to n do: /*所有硬币整数倍的均可以拼凑*/
        j:=0
        while(j*X[j]<=v):
            dp[j*X[j]]:=true
            j++
    for k:=1 to n do:
        for j:=0 to v do:
            if dp[j]=true:
                dp[j+X[i]]:=true
    return dp[v]
```

RepCoin.func

2) 与 problem 14.2 类似，取子问题为 $dp[i][j]$，含义为使用前 i 个数能否满足金额和为 j。对于 $dp[i][j]$ 要么使用了第 i 个元素，要么使用了前 i-1 个元素。故其递推式为:

$$dp[i][j] = (dp[i-1][j])||(dp[i][j - A[i]]).$$

3) 即找最少的硬币来兑换目标金额。取子问题 $dp[i]$ 为组成金额 i 所需最少的硬币数量，则对应的递推方程为:

$$dp[i] = \min(dp(i - x_k)) + 1, k = 0, 1, \cdots, n.$$

```
MinRepCoin(X,v):
    dp:=[v+1] /*dp数组大小为(v+1)的一维数组*/
    for i:=0 to v do: /*枚举0-v金额*/
        for j:=1 to n do: /*枚举使用每一枚硬币*/
            if X[j]<=i: /*硬币面值不超过所需金额*/
                dp[i]:=min(dp[i],dp[i-X[j]]+1)
    return dp[v]
```

MinRepCoin.func

problem 14.14

解 每个点设有两种状态：

$dp[i][0]$ 表示点 i 属于点覆盖集，并且以点 i 为根的子树中所连接的边都被覆盖的情况下，点覆盖集中所包含最少点的个数。

$dp[i][1]$ 表示点 i 不属于点覆盖集，并且以点 i 为根的子树中所连接的边都被覆盖的情况下，点覆盖集中所包含最少点的个数。

对于第一种状态，其最小个数等于所有子节点两种状态中最小值之和并加其本身。

对于第二种状态，因为点 i 所连接的边都应被覆盖，故点 i 的子节点都应被加入点覆盖集。故其最小个数等于所有子节点的第一种状态之和。

$$\begin{cases} dp[i][0] = \sum \min(dp[j][0], dp[j][1]), parent[j] = i, \\ dp[i][1] = \sum (dp[j][0]), parent[j] = i. \end{cases}$$

```
MiniVertexCover(u):
    dp[u][0]:=1;
    dp[u][1]:=0;
    for child v of u do:
        MiniVertexCover(v)
        dp[u][0]+=min(dp[v][0],dp[v][1])
        dp[u][1]+=dp[v][0]

MiniVertexCoverWrapper(T,root):
    dp:=[V+1][2]  /*dp数组大小为2(V+1)的一维数组*/
    dp(root)
    return min(dp[root][0],dp[root][1])
```

MiniVertexCover.func