卷积码Viterbi译码

1. 原理

1、卷积码：将k个信息比特变成n个比特，但k和n通常小，特别适合串行形式的传输，时延小。卷积码编码后的n个码元不仅与当前的k个信息有关，还与前面的N-1段信息有关，编码过程中互相关联的码元个数为n^N。卷积码的纠错性能随N的增加而增大，而差错率随N的增加而指数降低。卷积码编码，它常常与维特比译码结合使用。（2,1,3）卷积码编码器是最常见的卷积码编码器。

2、动态规划：动态规划程序设计是对解最优化问题的一种途径、一种方法，而不是一种特殊算法。不象搜索或数值计算那样，具有一个标准的数学表达式和明确清晰的解题方法。动态规划程序设计往往是针对一种最优化问题，由于各种问题的性质不同，确定最优解的条件也互不相同，因而动态规划的设计方法对不同的问题，有各具特色的解题方法，而不存在一种万能的动态规划算法，可以解决各类最优化问题。

3、Viterbi译码是根据接收序列在码的格图上找出一条与接收序列距离（或其他量度）为最小的一种算法。它和运筹学中求最短路径的算法相类似。若接收序列为R=(10100101100111)，译码器从某个状态,例如从状态ɑ出发,每次向右延伸一个分支（对于l＜L,从每个节点出发都有 2=2种可能的延伸,其中L是信息序列段数,对l≥L，只有一种可能）,并与接收数字相应分支进行比较,计算它们之间的距离，然后将计算所得距离加到被延伸路径的累积距离值中。对到达每个状态的各条路径（有2=2条）的距离累积值进行比较，保留距离值最小的一条路径，称为幸存路径（当有两条以上取最小值时，可任取其中之一）,译码过程如图。图中标出到达各级节点的幸存路径的距离累积值。对给定 R的估值序列为=(10111)。这

种算法所保留的路径与接收序列之间的似然概率为最大，所以又称为最大似然译码。这种译码的译码约束长度常为编码约束长度的数倍，因而可以纠正不多于(df/2)个错误。

1. 问题描述：

输入比特流个数、比特流和有噪信道的误码率（0~1）；对比特流数据进行补0操作（在比特流的最后编码器中仍保存着2个之前输入比特的状态，因此需要进行补0操作，即给输入比特流加上2个0比特）；对补0后的比特流进行（2,1,3）卷积码编码操作，编码输出的第一个结果是输入、第一个编码器存储的值和第二个编码器存储的值逻辑加操作的结果，第二个结果是输入和第二个编码器存储的值逻辑加操作的结果；对（2,1,3）卷积码编码输出的数据进行传输（加上误码）；

对从信道得到的有误码的比特流进行维特比译码：对比特流进行分组，2个一组循环；根据这2个比特对当前的4个状态（StateNode）计算从它出发到它可能到达的2个状态对应路径的汉明距离，并保存对应的译码序列和汉明距离；根据上一步的结果，取汉明距离小的更新这4个状态；最后，第1个状态（0状态）对应的译码序列就是维特比译码的结果（因为补零操作保证了最后肯定会回到0状态）。

1. 算法分析（时间复杂度及可能的改进）：

假设输入比特流序列的长度为L。由于（2,1,3）卷积码的状态数是4，对每个时刻要做4次“加-比-选”操作得到4个状态的残留路径，每次“加-比-选”操作包括2次加法和1次比较，因此总运算量约为4L次“加-比-选”操作。同时要能保存4条残留路径，因此需要4L个存贮单元。由此可见，（2,1,3）卷积码的维特比译码算法的时间和空间复杂度均与比特流序列长度呈线性关系，但维特比译码算法的时间空间复杂度与卷积码的约束长度呈指数关系。

可能的改进： 由于维特比译码算法的时间空间复杂度与卷积码的约束长度呈指数关系，因此对状态数很大的卷积码编码，维特比算法要经一定的修正后才可能实用，常用的算法是缩减状态的维特比译码，即在每一时刻，只处理部分的状态。

1. 算法实现：

int encode(unsigned int \*symbols, unsigned int \*data, unsigned int nbytes, unsigned int startstate)

{

unsigned int j;

unsigned int input,a1=0,a2=0,a3=0,a4=0,a5=0,a6=0;

for(j=0;j<nbytes;j++)

{

input=\*data;

data++;

\*symbols = input^a1^a2^a3^a6; //c1(171)

symbols++;

\*symbols = input^a2^a3^a5^a6; //c2(133)

symbols++;

a2=a1;

a1=input;

}

return 0;

}

void viterbi(int initialstate, int \*viterbiinput, int \*viterbioutput )

{

struct sta /\*定义网格图中每一点为一个结构体,其元素包括\*/

{

int met; /\*转移到此状态累计的度量值\*/

int value; /\*输入符号 \*/

struct sta \*last; /\*及指向前一个状态的指针\*/

};

struct sta state[4][N];

struct sta \*g,\*head;

int i,j,p,q,t,r,u,l;

for(i=0;i<4;i++)

{

for(j=0;j<N;j++)

state[i][j].met=0;

} /\* 初始化每个状态的度量值\*/

for(l=0;l<4;l++)

{

state[l][0].met=trandistance(\*viterbiinput,initialstate,l);

state[l][0].value=traninput(initialstate,l);

state[l][0].last=NULL;

}

viterbiinput++; /\*扩展第一步幸存路径\*/

for(t=1;t<N;t++)

{

for(p=0;p<4;p++)

{

state[p][t].met=state[0][t-1].met+trandistance(\*viterbiinput,0,p);

state[p][t].value=traninput(0,p);

state[p][t].last=&state[0][t-1];

for(q=0;q<4;q++)

{

if(state[q][t-1].met+trandistance(\*viterbiinput,q,p)<state[p][t].met)

{

state[p][t].met=state[q][t-1].met+trandistance(\*viterbiinput,q,p);

state[p][t].value=traninput(q,p);

state[p][t].last=&state[q][t-1];

}

}

}

viterbiinput++;

state[l][0].last=NULL;

}

viterbiinput++; /\*扩展第一步幸存路径\*/

for(t=1;t<N;t++)

{

for(p=0;p<4;p++)

{

state[p][t].met=state[0][t-1].met+trandistance(\*viterbiinput,0,p);

state[p][t].value=traninput(0,p);

state[p][t].last=&state[0][t-1];

for(q=0;q<4;q++)

{

if(state[q][t-1].met+trandistance(\*viterbiinput,q,p)<state[p][t].met)

{

state[p][t].met=state[q][t-1].met+trandistance(\*viterbiinput,q,p);

state[p][t].value=traninput(q,p);

state[p][t].last=&state[q][t-1];

}

}

}

viterbiinput++;

}/\*计算出剩余的幸存路径\*/

r=state[0][N-1].met; /\*找出n步后度量值最小的状态准备回溯路由\*/

g=&state[0][N-1];

for(u=N;u>0;u--) /\*向前递归的找出最大似然路径 \*/

{

\*(viterbioutput+(u-1))=g->value;

g=g->last;

} /\* for(u=0;u<8;u++) \*(viterbioutput+u)=state[u][2].met; \*/ /\*此行程序可用于检测第n列的 度量值\*/

}

1. 总结：

动态规划算法通常用于求解具有最优子结构的问题。在这类问题中，可能会有许多可行解，每一个解都对应于一个值，我们希望找到具有最优值的解。动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。若用分治法来解这类问题，有些子问题被重复计算多次。如果我们能够保存已解决的子问题的答案，而在需要时再找出已求得的答案，这样就可以避免大量的重复计算，节省时间。

由于没有实际建立卷积码与Viterbi译码的概念，以及对于动态规划算法尚未完全掌握，所以在理解题目与实验过程中遇到了很多困难，通过上网查资料进行学习对其有了大致的了解，知道了算法主要应该包括补0、卷积编码、信道传输和维特比译码4步操作，并实现该算法。通过实现该算法，我学习了在实际中应用动态规划算法。

1. 源码：见附录

#include<stdio.h>

#include "Conio.h"

#define N 7

#include "math.h"

#include <stdlib.h>

#include<time.h>

#define randomize() srand((unsigned)time(NULL))

int encode(unsigned int \*symbols, /\*编码输出\*/unsigned int \*data, /\*编码输入\*/unsigned int nbytes, /\*nbytes=n/16,n为实际输入码字的数目\*/unsigned int startstate /\*定义初始化状态\*/)

{

unsigned int j;

unsigned int input,a1=0,a2=0,a3=0,a4=0,a5=0,a6=0;

for(j=0;j<nbytes;j++)

{

input=\*data;

data++;

\*symbols = input^a1^a2^a3^a6; //c1(171)

symbols++;

\*symbols = input^a2^a3^a5^a6; //c2(133)

symbols++;

a2=a1;

a1=input;

}

return 0;

}

int trandistance(int m, int state1, int state2)

/\*符号m与从state1到state2时输出符号的汉明距离,如果state1无法到state2则输出度量

值为100\*/

{

int sym,sym1,sym2;

int c;

sym1=((state2>>1)&1)^(state2&1)^(state1&1);

sym2=((state2>>1)&1)^(state1&1);

sym=(sym1<<1) | sym2;

if ( ((state1&2)>>1)==(state2&1))

c=((m&1)^(sym&1))+(((m>> 1)&1)^((sym >> 1)&1));

else

c=10000;

return c;

}

int traninput(int a,int b) /\*状态从a到b时输入卷积码的符号\*/

{

int c;

c=((b&2)>>1);

return c;

}

int tranoutput(int a,int b) /\*状态从a到b时卷积码输出的符号\*/

{

int c,s1,s2;

s1=(a&1)^((a&2)>>1)^((b&2)>>1);

s2=(a&1)^((b&2)>>1);

c=(s1<<1)|s2;

return c;

}

void viterbi(int initialstate, /\*定义解码器初始状态\*/int \*viterbiinput, /\*解码器输入码字序列\*/int \*viterbioutput /\*解码器输出码字序列\*/ )

{

struct sta /\*定义网格图中每一点为一个结构体,其元素包括\*/

{

int met; /\*转移到此状态累计的度量值\*/

int value; /\*输入符号 \*/

struct sta \*last; /\*及指向前一个状态的指针\*/

};

struct sta state[4][N];

struct sta \*g,\*head;

int i,j,p,q,t,r,u,l;

for(i=0;i<4;i++)

{

for(j=0;j<N;j++)

state[i][j].met=0;

} /\* 初始化每个状态的度量值\*/

for(l=0;l<4;l++)

{

state[l][0].met=trandistance(\*viterbiinput,initialstate,l);

state[l][0].value=traninput(initialstate,l);

state[l][0].last=NULL;

}

viterbiinput++; /\*扩展第一步幸存路径\*/

for(t=1;t<N;t++)

{

for(p=0;p<4;p++)

{

state[p][t].met=state[0][t-1].met+trandistance(\*viterbiinput,0,p);

state[p][t].value=traninput(0,p);

state[p][t].last=&state[0][t-1];

for(q=0;q<4;q++)

{

if(state[q][t-1].met+trandistance(\*viterbiinput,q,p)<state[p][t].met)

{

state[p][t].met=state[q][t-1].met+trandistance(\*viterbiinput,q,p);

state[p][t].value=traninput(q,p);

state[p][t].last=&state[q][t-1];

}

}

}

viterbiinput++;

state[l][0].last=NULL;

}

viterbiinput++; /\*扩展第一步幸存路径\*/

for(t=1;t<N;t++)

{

for(p=0;p<4;p++)

{

state[p][t].met=state[0][t-1].met+trandistance(\*viterbiinput,0,p);

state[p][t].value=traninput(0,p);

state[p][t].last=&state[0][t-1];

for(q=0;q<4;q++)

{

if(state[q][t-1].met+trandistance(\*viterbiinput,q,p)<state[p][t].met)

{

state[p][t].met=state[q][t-1].met+trandistance(\*viterbiinput,q,p);

state[p][t].value=traninput(q,p);

state[p][t].last=&state[q][t-1];

}

}

}

viterbiinput++;

}/\*计算出剩余的幸存路径\*/

r=state[0][N-1].met; /\*找出n步后度量值最小的状态准备回溯路由\*/

g=&state[0][N-1];

for(u=N;u>0;u--) /\*向前递归的找出最大似然路径 \*/

{

\*(viterbioutput+(u-1))=g->value;

g=g->last;

} /\* for(u=0;u<8;u++) \*(viterbioutput+u)=state[u][2].met; \*/ /\*此行程序可用于检测第n列的 度量值\*/

}

void decode(unsigned int \*input, int \*output,int n)

{

int viterbiinput[100];

int j;

for(j=0;j<n+2;j++)

{

viterbiinput[j]=(input[j\*2]<<1)|input[j\*2+1];

}

viterbi(0,viterbiinput,output);

}

int main()

{

unsigned int encodeinput[100],wrong[10]={0,0,0,0,0,0,0,0,0,0},encodeoutput[100];

int n=5,i,m,j=0,decodeinput[100],decodeoutput[100];

randomize();

for(i=0; i<n; i++)

encodeinput[i]=rand()%2;

encodeinput[n]= encodeinput[n+1]=0;

encode(encodeoutput,encodeinput,n+2,0);

printf("the input of encoder is :\n"); //信息源输入的信息码（随机产生）

for(i=0;i<n; i++) printf("%2d",encodeinput[i]);

printf("\n");

printf("the output of encoder is :\n"); //编码之后产生的卷积码

for(i=0;i<(n+2)\*2;i++)

{

printf("%2d",encodeoutput[i]);

if(i%20==19) printf("\n");

} printf("\n");

system("PAUSE") ;

}