**基于2-1-3卷积码的**

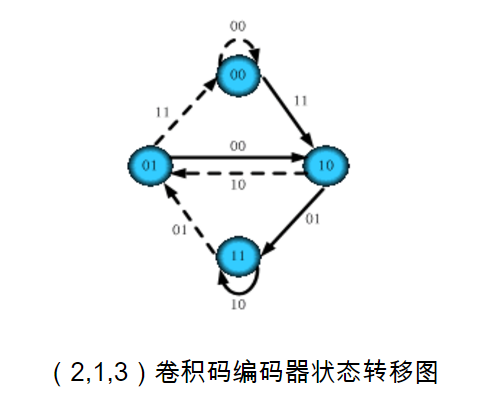
**Viterbi译码算法实验报告**

1. **算法原理**

本实验完成了基于（2,1,3）卷积码的译码与纠错。Viterbi算法的本质是最大似然原则。N位输入经过卷积输出后，得到2N位的卷积码，这2N位中，每两位与原码的码字对应。算法基于这样的一个准则：求解出一条状态转移路径，经过这条路径，可以使某种权重最小。根据题设要求，显然这个权重是与收到的卷积码的相似度，这里我们用两个码字之间的海明距离来衡量它们是否相似。也就是说，这个算法找到了一个与收到的卷积码最相似的一条路径，这个路径的输出就是算法的译码输出。

当然路径本身是有约束的，我们用层级来表示每一个阶段选择的译码输出。每一层均有4个状态，他们是D00，D01，D10，D11。对于一条边的输出，它们也只可能是Z00，Z01，Z10和Z11。本质上来说，（2-1-3）卷积码的产生是一个FSM模型，一个状态的后继状态被输入唯一确定。

这个FSM可以表现为如下的状态转移图：



注意，虚线代表输入0，实线代表输入1。

我们可以想象这样一个矩阵，矩阵行数为4，每一行代表了自动机的一个状态，每一列代表了一个译码时序。每更新一列，每个状态的当前权重都需要进行更新（选权重和最小的）。显然，如果想得到全部的信息，矩阵中的每一个单元还需要加入额外的标记，以满足动态规划的需要。

经过上述描述，可以容易地看出这类似于有限制的最短路径问题，只不过权重被海明距离所表达。当前层级状态的最短路径可以这样被找到：计算上一层级（前面提到的那个矩阵的上一列）到当前层级每一状态可能的连接，这些连接都带有一个权重。当前候选最短路径可以这样被表达：边的权重与到达上一状态的权重之和。当然我们要从这些后选中找到最短路径，那么还有比较、选择的过程。

这个问题的本质变为，在这些层级的连线构成的有向图中，找到一条最短的路径使得权重（总海明距离）最小。

1. **问题描述**

本实验要解决的问题是：给定一组接受到的卷积码字，通过viterbi算法译码，并纠错。请注意：如果没有任何错误，那么输入和输出码字应该是同一码字。随后这个码字被FSM译码。我们只需要得到纠错之后的卷积码序列，还原卷积码序列（从卷积码到输入码的过程）十分简单，这些信息在上述提到的矩阵单元中被存储。

1. **算法过程分析**

算法构造一个矩阵mat来模拟寻找路径的过程，矩阵的每一行代表一个状态，每一列代表一个译码时间点。矩阵的列数被接收到的码字长度唯一确定。

算法的关键是计算权重、比较权重、更新权重的过程，这个过程在crr函数中实现，采用的是有限状态枚举方法，因为考虑到状态转移模式并不多（一个状态的前驱状态有且只有两个，这是由（2-1-3）卷积码的编码过程决定的）。

因此，算法实现的过程就是填写mat矩阵的过程，当mat矩阵根据bit流填写完成后，只需要找到最后一列的最小权重状态（终态），再回溯至初态即可得到路径序列，这个路径序列的反向输出就是译码结果。

算法流程图可以简单的表示如下：

1. **算法复杂度分析**

假设共有N级运算。

对于穷举法而言，每一级有4个待选元，共需选择N次，总共的穷举数量不少于4^N，因此算法复杂度为O（4^N）。

本实验使用了动态规划算法，使得更新在每一级产生，计算第N级时只需要依据N-1级的数据即可。注意到在（2-1-3）卷积码对应的FSM限制下，每一级至多要进行8次比较，供需比较N-1次，因此算法的时间复杂度为O（N）。这表明，在这种情况下，viterbi算法的时间复杂度是线性的。大体上来看，只需要从第一列到最后一列依次填写好mat矩阵的内容即可。

对于空间复杂度，主要的开销在于mat矩阵。我们已经分析过，矩阵的列数是与输入的比特流长度有关，而行数是FSM对应的可能的状态集。即空间复杂度为O（N）。

1. **可能的改进**

本实验的算法在下面几个部分仍存在缺陷

1. **纠错的正确性问题：**请注意，（2-1-3）卷积码的纠错能力是有限的，在连续的3个连续码流中纠正1bit的错误，可以看出虽然算法无论如何都会运行，但纠错结果未必是正确的。
2. **算法不具有一般性：**本算法仅针对2-1-3卷积码进行译码和纠错，不属于此类的卷积码不能得到正确的输出。 这里引发了我的思考：是否viterbi算法存在一般的形式，对于这种链式结构，或者说对于任意的（n,k,m）卷积码都可以进行译码。答案是肯定的，但本次实验不讨论通用的viterbi算法。
3. **算法实现**
4. **数据结构和函数**

enum state { Z00 , Z01 , Z10 , Z11 }; // 寄存器状态集

typedef enum state STATE;

enum data { D00 , D01 , D10 ,D11 }; // 数据集 （2,1,3）卷积码 是{0,1} -> {DATA} 的映射

typedef enum data DATA;

struct path\_node

{

STATE last\_state; // 上一个状态

int total\_weight; // 当前权重

DATA data\_chose; // 最大似然选择

int bit\_chose; // 译码选择

};

typedef struct path\_node PATH\_NODE;

这里的STATE是枚举类型，表示了寄存器所有可能的状态，即所有可能的译码单元。

这里的DATA是枚举类型，表示可所有可能的数据集合。

PATH\_NODE是算法的核心结构，是mat矩阵的单元。last\_state记录了到达该状态之前的状态，total\_weight记录了对应当前状态下的最小权重，data\_chose是路径信息，是动态规划的附属产物，bit\_chose是译码结果的标记，这个标记被data\_chose和last\_state所确定（本质上是模拟FSM）。

void init\_mat(int input[]) // 设置寄存器初始的状态为00

int ham(DATA d1 , DATA d2) // 由海明矩阵快速计算海明距离

void crr(int input[] , int len , int &num\_t , STATE &end\_state) // viterbi译码过程

void make\_output(int ed , STATE end\_state) // 倒序构建输出序列

1. **算法核心流程**

重点介绍crr函数的实现过程，即viterbi的译码过程。

填写mat的本质是路径的选择，这里我们枚举了所有可能的路径，抓取其中一个路径选择单元为例：

if( ham(bi\_data,D00) + mat[Z00][t-1].total\_weight < ham(bi\_data,D11) + mat[Z01][t-1].total\_weight ) // 枚举Z00的前一状态，并计算当前最小权重， 记录路径选择

{

mat[Z00][t].total\_weight = ham(bi\_data,D00) + mat[Z00][t-1].total\_weight; // 两条可能的路径选权重最小的

mat[Z00][t].last\_state = Z00; // 记录路径

mat[Z00][t].data\_chose = D00; // 记录状态选择

mat[Z00][t].bit\_chose = 0; // 记录译码选择

}

else

{

mat[Z00][t].total\_weight = ham(bi\_data,D11) + mat[Z01][t-1].total\_weight;

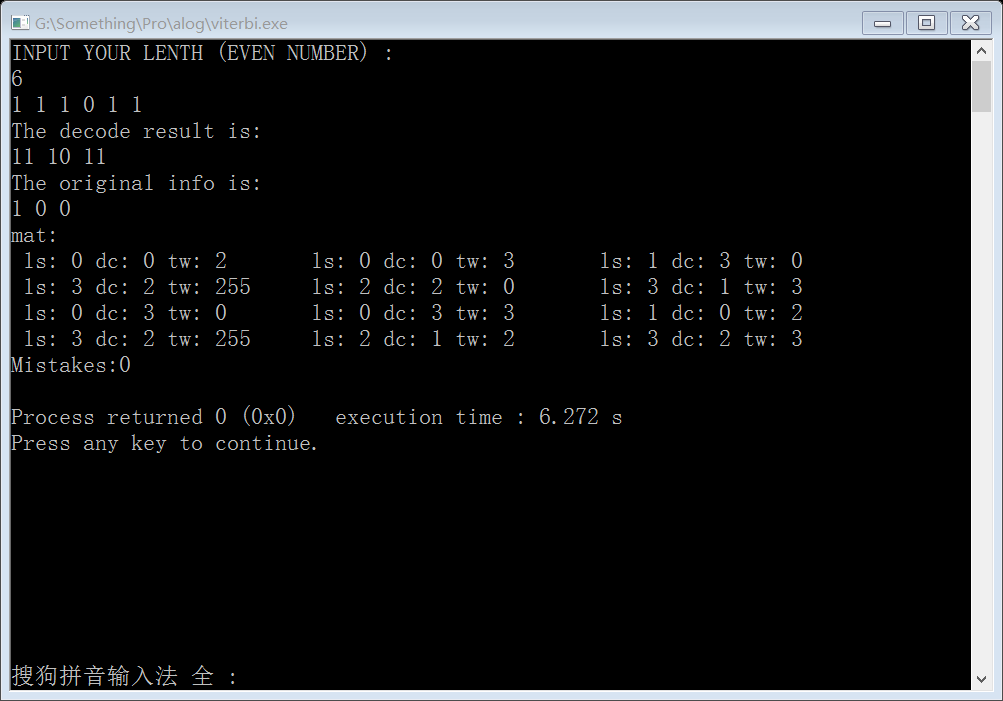
mat[Z00][t].last\_state = Z01;

mat[Z00][t].data\_chose = D11;

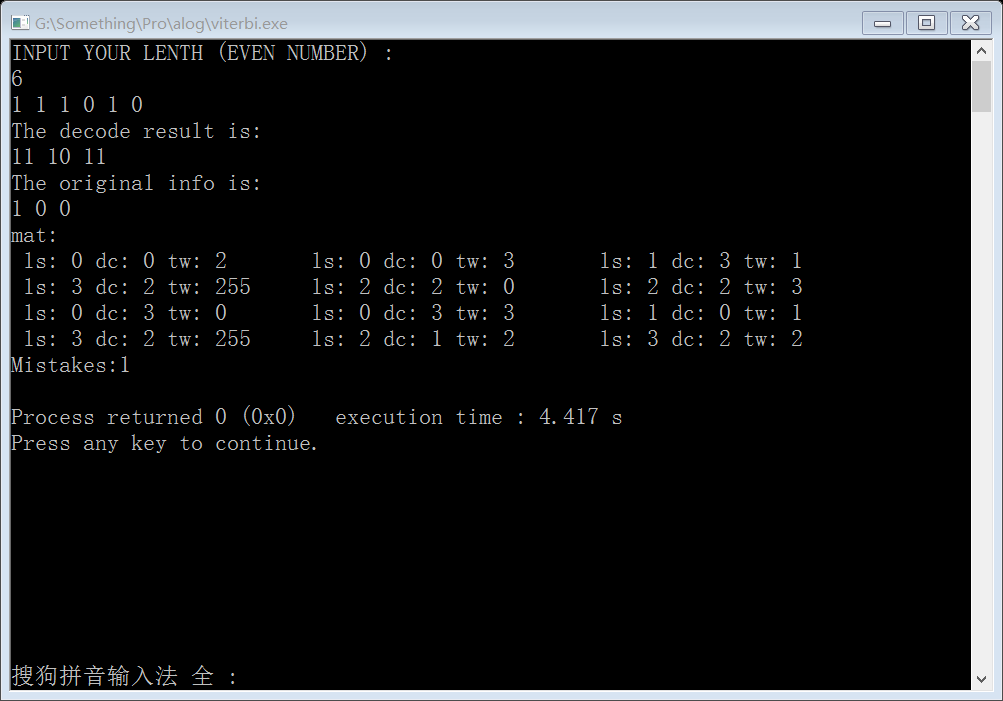
mat[Z00][t].bit\_chose = 0;

}

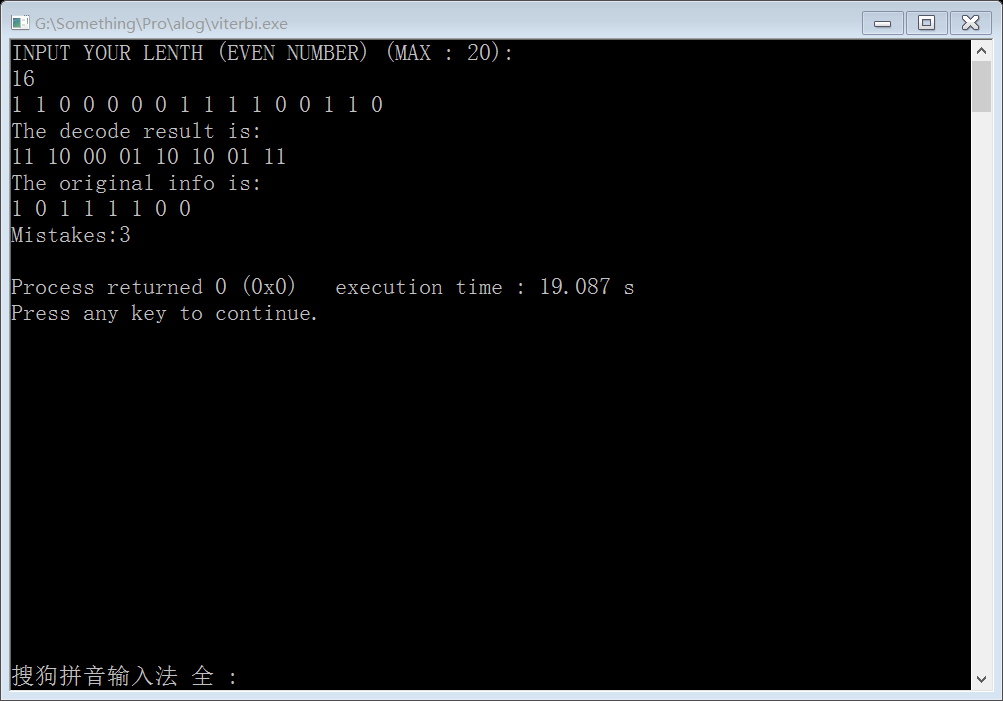
我们现在要填写的是译码时序t下的Z00状态单元，由FSM我们不难看出，Z00的前驱状态只可能是Z01和Z00，那么计算来自于这两个前驱的路径权重和，选择小的一条，记录信息，就完成了该单元的填写，其余单元的填写过程以此类推。3、 **输出结果示例与说明**



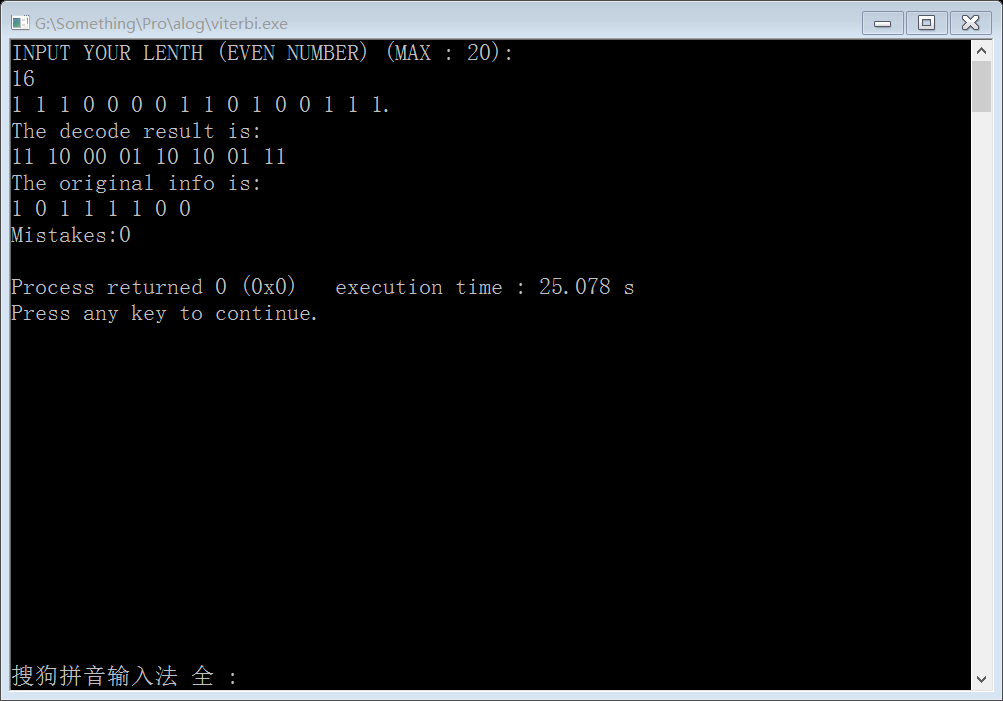
这里接受的卷积码为长度为6的bit流，经过算法输出，没有错误，原始信息为 1 0 0。在接下来的情况，我们人为地在这个bit流中制造1bit错误，使接受序列变为 1 1 1 0 1 0。



可以看出，算法正确地对错误序列进行纠错，并指出错误数为1bit。纠错之后的结果为 11 10 11，这与上一个测试结果相符。



当序列稍长时，算法的适应性良好。11 10 00 01 10 10 01 11作为下一次的输入，理论上应该得到无误码的判断结果。



实验结果与预测的相符，说明程序正确运行。

1. **总结和反思**

本次编码依然是陌生的算法：viterbi算法，好在其特殊的形式，对于2-1-3这种最简单的卷积码来说不算复杂。它有点类似于有限制的最短路径问题，只不过这个问题被搬移到一个FSM上。

其实，最重要的是这次实验强化了我的自学能力。你有可能第一次看不出它是一个动态规划问题，但是经过了大量的资料查阅和参考，当你搞懂算法本质的时候，可以看出逐级递进的关系和内涵。某译码时序的输出可以通过前一译码时序的结果得到，类似于这样的递推、记录和最后的回溯过程，是典型的动态规划问题。

需要说明的是，由于理论部分借鉴了互联网的资料，本实验报告的算法原理部分非原创，除此之外均为独立完成。

#include<iostream>

#define MAX\_WIDTH 4

#define MAX\_INPUT 20

#define MAX\_WEIGHT 255

using namespace std;

enum state { Z00 , Z01 , Z10 , Z11 }; // 寄存器状态集

typedef enum state STATE;

enum data { D00 , D01 , D10 ,D11 }; // 数据集 （2,1,3）卷积码 是{0,1} -> {DATA} 的映射

typedef enum data DATA;

struct path\_node

{

STATE last\_state; // 上一个状态

int total\_weight; // 当前权重

DATA data\_chose; // 最大似然选择

int bit\_chose; // 译码选择

};

typedef struct path\_node PATH\_NODE;

PATH\_NODE mat[MAX\_WIDTH][MAX\_INPUT]; // 状态转移矩阵

PATH\_NODE output[MAX\_INPUT]; // 输出路径

DATA d\_input[MAX\_INPUT/2]; // DATA形式的纠错数组

int ham\_mat[MAX\_WIDTH][MAX\_WIDTH] = {0,1,1,2,

1,0,2,1,

1,2,0,1,

2,1,1,0}; // 数据集合的海明矩阵

void init\_mat(int input[]) // 设置寄存器初始的状态为00

{

mat[1][0] = {Z11,MAX\_WEIGHT,D10,0};

mat[3][0] = {Z11,MAX\_WEIGHT,D10,0};

if(input[0] == 1 && input[1] == 1)

{

mat[0][0] = {Z00,2,D00,0};

mat[2][0] = {Z00,0,D11,1};

}

else if((input[0] == 1 && input[1] == 0) ||

(input[0] == 0 && input[1] == 1))

{

mat[0][0] = {Z00,1,D00,0};

mat[2][0] = {Z00,1,D11,1};

}

else

{

mat[0][0] = {Z00,0,D00,0};

mat[2][0] = {Z00,2,D11,1};

}

}

int ham(DATA d1 , DATA d2) // 由海明矩阵快速计算海明距离

{

return ham\_mat[d1][d2];

}

DATA int\_to\_bi(int i,int j)

{

if(i==1&&j==1)

return D11;

if(i==1&&j==0)

return D10;

if(i==0&&j==1)

return D01;

return D00;

}

void crr(int input[] , int len , int &num\_t , STATE &end\_state) // viterbi译码过程

{

int t;

DATA bi\_data;

int mini\_weight = MAX\_WEIGHT;

for(int i = 2 ; i<=len-1;i+=2 )

{

t = i/2;

bi\_data = int\_to\_bi(input[i],input[i+1]);

//cout<<"bidata:"<<bi\_data<<endl;

if( ham(bi\_data,D00) + mat[Z00][t-1].total\_weight < ham(bi\_data,D11) + mat[Z01][t-1].total\_weight ) // 枚举Z00的前一状态，并计算当前最小权重，记录路径选择

{

mat[Z00][t].total\_weight = ham(bi\_data,D00) + mat[Z00][t-1].total\_weight; // 两条可能的路径选权重最小的

mat[Z00][t].last\_state = Z00; // 记录路径

mat[Z00][t].data\_chose = D00; // 记录状态选择

mat[Z00][t].bit\_chose = 0; // 记录译码选择

}

else

{

mat[Z00][t].total\_weight = ham(bi\_data,D11) + mat[Z01][t-1].total\_weight;

mat[Z00][t].last\_state = Z01;

mat[Z00][t].data\_chose = D11;

mat[Z00][t].bit\_chose = 0;

}

if( ham(bi\_data,D10) + mat[Z10][t-1].total\_weight < ham(bi\_data,D01) + mat[Z11][t-1].total\_weight ) // 同上，枚举Z01

{

mat[Z01][t].total\_weight = ham(bi\_data,D10) + mat[Z10][t-1].total\_weight;

mat[Z01][t].last\_state = Z10;

mat[Z01][t].data\_chose = D10;

mat[Z01][t].bit\_chose = 0;

}

else

{

mat[Z01][t].total\_weight = ham(bi\_data,D01) + mat[Z11][t-1].total\_weight;

mat[Z01][t].last\_state = Z11;

mat[Z01][t].data\_chose = D01;

mat[Z01][t].bit\_chose = 0;

}

if( ham(bi\_data,D11) + mat[Z00][t-1].total\_weight < ham(bi\_data,D00) + mat[Z01][t-1].total\_weight ) // 同上，枚举Z10

{

mat[Z10][t].total\_weight = ham(bi\_data,D11) + mat[Z00][t-1].total\_weight;

mat[Z10][t].last\_state = Z00;

mat[Z10][t].data\_chose = D11;

mat[Z10][t].bit\_chose = 1;

}

else

{

mat[Z10][t].total\_weight = ham(bi\_data,D00) + mat[Z01][t-1].total\_weight;

mat[Z10][t].last\_state = Z01;

mat[Z10][t].data\_chose = D00;

mat[Z10][t].bit\_chose = 1;

}

if( ham(bi\_data,D01) + mat[Z10][t-1].total\_weight < ham(bi\_data,D10) + mat[Z11][t-1].total\_weight ) // 同上，枚举Z11

{

mat[Z11][t].total\_weight = ham(bi\_data,D01) + mat[Z10][t-1].total\_weight;

mat[Z11][t].last\_state = Z10;

mat[Z11][t].data\_chose = D01;

mat[Z11][t].bit\_chose = 1;

}

else

{

mat[Z11][t].total\_weight = ham(bi\_data,D10) + mat[Z11][t-1].total\_weight;

mat[Z11][t].last\_state = Z11;

mat[Z11][t].data\_chose = D10;

mat[Z11][t].bit\_chose = 1;

}

}

num\_t = t;

//cout<<"t"<<t<<endl;

for(int i = 0 ;i <=MAX\_WIDTH -1 ;i++)

{

//cout<<mat[i][t].total\_weight<<" ";

if(mat[i][t].total\_weight < mini\_weight)

{

end\_state = (STATE)i;

mini\_weight = mat[i][t].total\_weight;

}

}

//cout<<"end\_s: "<<end\_state<<endl;

}

void make\_output(int ed , STATE end\_state) // 倒序构建输出序列

{

STATE last\_s;

output[ed].data\_chose = mat[end\_state][ed].data\_chose;

output[ed].bit\_chose = mat[end\_state][ed].bit\_chose;

last\_s = mat[end\_state][ed].last\_state;

for(int t = ed - 1 ; t>=0 ; t--)

{

output[t].data\_chose = mat[last\_s][t].data\_chose;

output[t].bit\_chose = mat[last\_s][t].bit\_chose;

//cout<<"output ["<<t<<"]:"<<output[t]<<" ";

last\_s = mat[last\_s][t].last\_state;

//cout<<"last\_s:"<<last\_s<<endl;

}

}

void print\_output(int ed) // 打印并输出译码结果

{

cout<<"The decode result is: "<<endl;

for(int i = 0;i<=ed;i++)

{

if(output[i].data\_chose == D00)

cout<<"00 ";

else if(output[i].data\_chose == D01)

cout<<"01 ";

else if(output[i].data\_chose == D10)

cout<<"10 ";

else cout<<"11 ";

}

cout<<endl<<"The original info is: "<<endl;

for(int i = 0;i<=ed;i++)

{

cout<<output[i].bit\_chose<<" ";

}

cout<<endl;

}

void print\_mat(int num)

{

cout<<"mat:"<<endl;

for(int row = 0;row<=MAX\_WIDTH - 1;row++ )

{

for(int col = 0;col<=num;col++)

{

cout<<" ls: "<<mat[row][col].last\_state

<<" dc: "<<mat[row][col].data\_chose

<<" tw: "<<mat[row][col].total\_weight<<"\t";

}

cout<<endl;

}

}

void make\_input(int input[],int len) // 数码结构的转换，方便输出

{

int p;

for(int i = 0;i<=len-1;i+=2)

{

p = i/2;

d\_input[p] = int\_to\_bi(input[i],input[i+1]);

}

}

int count\_err(int num) // 计算误码数量

{

int err = 0;

for(int i = 0;i<=num;i++)

err += ham(d\_input[i],output[i].data\_chose);

return err;

}

int main(void)

{

int len;

int input[MAX\_INPUT];

int err;

int num;

STATE end\_s;

cout<<"INPUT YOUR LENTH (EVEN NUMBER) (MAX : 20): "<<endl;

cin>>len;

for(int i = 0;i<=len-1;i++)

cin>>input[i];

init\_mat(input);

crr(input,len,num,end\_s); // 最大似然原则，viterbi算法，构造mat矩阵（路径集合）

make\_output(num,end\_s); // 构建输出路径

make\_input(input,len); // 为统计误码做准备

print\_output(num);

//print\_mat(num);

err = count\_err(num); // 统计误码过程

cout<<"Mistakes:"<<err<<endl;

return 0;

}