密码分析学

第 2 次大作业

第六组

2021 年 10 月 17 日

目录

摘要

组员分工

王文凯 201900401024：task1,6 的 python 代码实现，task6 报告书写

初婧雯 201900460011：差分算法原理报告书写

周睿泽 201900460023：task1,2,4 的 python 代码实现，部分原理及 task2,4 报告书写，报告整理

林若妍 201900460027：task1,2,3 的 python 代码实现，task1,3 报告书写

张自平 201900460035：task1,2,3,4 的 C++ 代码实现，task5 的 python 代码实现，task5 报告书写，报告校对

作业要求

1）编程实现 4 轮的 CipherFour 算法 (注意最后一轮有 P 置换)。

2）随机选取 20 组不同的密钥（例如（此处数据均按 16 进制），某一组取值 5b92, 064b, 1e03, a55f, ecbd），遍历 216 种 16 比特的输入对 $(x\ x \oplus 00f0)$，输入 4 轮的 CipherFour 算法，得到相应的输出对，统计输出差分出现的频率。将选择输入对的差分改为 0020，重复此实验，统计输出差分出现的频率。

3）尝试根据算法及 S 盒差分分布表计算差分 $(0,0,2,0) \xrightarrow{4R} (0,0,2,0)$ 的概率并与 2）的实验结果进行比较。

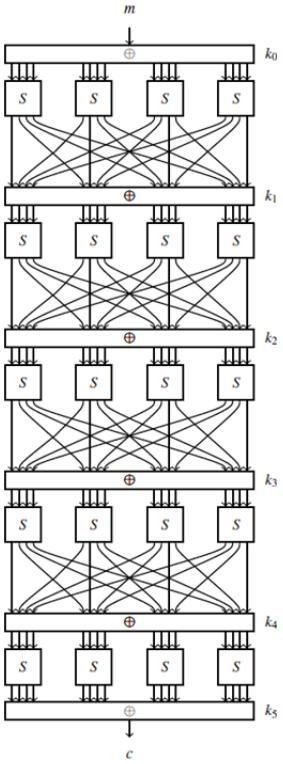
4）利用现成的随机数生成算法，生成 216 对 16 比特的随机数（相当于随机置换的输出对），统计异或值出现的频率并与 2）中结果进行比较。

5）能否找到 4 轮 CipherFour 算法的最优差分特征？最优差分呢？

6）根据压缩包中各组的数据（文件夹名即是组号，数据为固定密钥下 5 轮 CipherFour 算法的全部明密文对，plaintext_ 组号.txt 为所有明文，cipher_ 组号.txt 为所有密文，明密文按顺序对应。实际数据格式请按自己选用编程语言做适当修改。注意最后一轮无 P 置换），借助差分分析，实现 5 轮 CipherFour 算法的密钥恢复攻击。分别测试选择明文量 100 对和 500 对时的成功率（多次并行求解，计算求对的概率）。

1 实验原理

1.1 CipherFour 算法原理



4 轮 CIPHERFOUR 算法包括四轮的轮密钥异或、S 盒置换和 P 置换，其中最后一轮还需通过一次白化密钥异或输出，即为最终输出密文结果。

CIPHERFOUR 算法首先输入 16-bit 的明文，与 16-bit 的 k0 异或，得到的密钥加密后的结果。

加密后的 16-bit 按顺序分为四组 4 位二进制数字，即为 4 个十六进制数字，依次通过 S 盒（非线性变换）

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$S(x)$	6	4	c	5	0	7	2	e	1	f	3	d	8	a	9	b

得到 S 置换后的结果后由 P 置换将输出比特扩散到其他位置。即进行如下线性变换：

$$P_{i,j} = S_{j,i}$$

(其中，$S_{j,i}$ 为通过 S 置换后的第 j 组第 i 个输出，$P_{i,j}$ 为通过 P 置换后第 i 组第 j 个输出)

此为一轮加密（除最后一轮）。

最后一轮加密包括同上的轮密钥异或、S 盒置换和 P 置换（本题中最后一轮有 P 置换）以及一次与 16-bitk_5 进行的异或。注意这里轮密钥与白化密钥（k_5）均是独立随机选择的。

1.2 差分算法原理

差分分析的主要思想是通过选择满足特定差分的明文对，追踪差分在多轮加密过程中的变换情况，选择长轮数高概率差分。因为在差分分析的步骤中对于 AK、P 置换、E 扩展、MC、SR 均为线性变换，因此输入的差分值确定，那么输出的差分值也是确定的，即概率为 1。所以我们现在只需要考虑非线性部件 S 盒的差分分布。

设 m、n N，从 F_{2m} 到 F_{2n} 的非线性映射，也记为 S：$F_{2m} \to F_{2m}$，给 α F_{2m}，β F_{2n}，那么定义 $INs = x\ F_{2m}; S(x) \oplus S(x \oplus \alpha) = \beta$ 。对于 S 盒，输入对差分是 α，相应输出对的差分是 β，则 $\alpha \to \beta$ 是 S 盒的一个差分特征，且

$$Pr(\alpha \to \beta) = \frac{INs}{2^m}$$

值得注意的是 $Pr(0 \to 0) = 1$，且对于随机置换

$$Pr(\alpha \to \beta) = \frac{1}{2^n}$$

根据上面原理分别找 CIPHERFOUR 算法一轮差分特征、二轮差分特征、r 轮差分特征。

一轮差分特征

S 盒：令 $Pr(\alpha i \to\ i) = Pi$，则 $Pr((\alpha 1\ \alpha 2\ \alpha 3\ \alpha 4) \to (\beta 1\ \beta 2\ \beta 3\ \beta 4)) = P1P2P3P4$

P 置换：$Pr((\beta 1\ \beta 2\ \beta 3\ \beta 4) \to (1\ 2\ 3\ 4)) = 1$

因此一轮加密 $Pr((\alpha 1\ \alpha 2\ \alpha 3\ \alpha 4) \to\ 1\ 2\ 3\ 4)) = P1P2P3P4$

为了让我们计算方便，我们可以选择输入差分不全是 0 的情况，但是最多也只能是三个字节差分为 0，这样就只与一个字节的输入差分对应输出差分概率有关，而我们找最高概率那个即可。根据下面这个差分分布表可以看出

$f \to d$ 的概率最大，为 $\frac{10}{16}$，因此可以认为**一轮差分特征为** $(0,0,0,f) \to (1,1,0,1)$

$$Pr((0,0,0,f) \to (1,1,0,1)) = \frac{10}{16}$$

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	6	-	-	-	-	2	-	2	-	-	2	-	4	-
2	-	6	6	-	-	-	-	-	-	2	2	-	-	-	-	-
3	-	-	-	6	-	2	-	-	2	-	-	-	4	-	2	-
4	-	-	-	2	-	2	4	-	-	2	2	2	-	-	2	-
5	-	2	2	-	4	-	-	4	2	-	-	2	-	-	-	-
6	-	-	2	-	4	-	-	2	2	-	2	2	2	-	-	-
7	-	-	-	-	-	4	4	-	2	2	2	2	-	-	-	-
8	-	-	-	-	-	2	-	2	4	-	-	4	-	2	-	2
9	-	2	-	-	-	2	2	2	-	4	2	-	-	-	-	2
a	-	-	-	-	2	2	-	-	-	4	4	-	2	2	-	-
b	-	-	-	2	2	-	2	2	2	-	-	4	-	-	2	-
c	-	4	-	2	-	2	-	-	2	-	-	-	-	-	6	-
d	-	-	-	-	-	-	2	2	-	-	-	-	6	2	-	4
e	-	2	-	4	2	-	-	-	-	-	2	-	-	-	-	6
f	-	-	-	-	2	-	2	-	-	-	-	-	-	10	-	2

二轮差分特征

对于二轮差分特征，依旧是查看差分分布表，但是要注意上一轮的输出差分等于这轮的输入差分。接着上面的计算第二轮加密，上一轮的输出为 $(1,1,0,1)$。

S 盒：找概率最大的，可以发现是 $1 \to 2$，因此 $Pr((1,1,0,1) \to (2,2,0,2)) = (\frac{3}{16})^3$

P 置换：$Pr((2,2,0,2) \to (0,0,d,0) = 1$

那么对于二轮加密的差分特征为 $(0,0,0,f) \to (1,1,0,1) \to (0,0,d,0)$，且

$$Pr((0,0,0,f) \to (1,1,0,1) \to (0,0,d,0)) = (\frac{3}{16})^3 \times \frac{10}{16} = 0.033$$

因为影响概率的因素有两个，一个是 $\triangle in$ 和 $\triangle out$ 对应的概率，一个是 $\triangle in \neq 0$ 的 S 盒的个数（即活跃 S 盒个数），因此如果控制两轮 S 盒都只有一个 S 盒 $\triangle in \neq 0$，那么概率将会提高。因此我们可以发现上面差分并不是最优差分特征。为了找到最优差分特征，我们控制第一轮 S 盒输出差分的个数为 1。那么第一轮 S 盒的 $\triangle out$ 情况有四种，分别为 1,2,4,8，经过查找差分分布表可以发现最大值是 6，也就是说 $1 \to 2/2 \to 2/2 \to 1$。以 $2 \to 2$ 为例：

一轮：$(0,0,2,0)S \to (0,0,2,0)P \to (0,0,2,0)$ $P = \frac{6}{16}$

二轮：$(0,0,2,0)R \to (0,0,2,0)P \to (0,0,2,0)$ $P = (\frac{6}{16})^2 > (\frac{3}{16})^3 * \frac{10}{16}$

这样我们找到了**二轮加密的最优差分特征，即** $(0,0,2,0) \to (0,0,2,0) \to (0,0,2,0)$。

同时我们也可以同理推出 **r 轮的最优差分特征** $(0,0,2,0) \to (0,0,2,0) \to .... \to (0,0,2,0)$

$$Pr((0,0,2,0) \to (0,0,2,0) \to .... \to (0,0,2,0)) = (\frac{6}{16})^r$$

我们可以发现随着轮数的增加，对应的单条差分特征概率降低，将会趋于 $(\frac{1}{16})^4$

而**对于没有高概率的差分特征，我们只考虑头部和尾部，不考虑中间状态**。以四轮加密为例，头部（0,0,2,0），尾部（0,0,2,0），根据差分分布表至少可以找到以下四条路线，每条路线的概率均为 $(\frac{6}{16})^4$。

$$(0,0,2,0) \to (0,0,2,0) \to (0,0,2,0) \to (0,0,2,0) \to (0,0,2,0)$$

$$(0,0,2,0) \to (0,0,0,2) \to (0,0,0,1) \to (0,0,1,0) \to (0,0,2,0)$$

$$(0,0,2,0) \to (0,0,0,2) \to (0,0,1,0) \to (0,0,2,0) \to (0,0,2,0)$$

$$(0,0,2,0) \to (0,0,2,0) \to (0,0,0,2) \to (0,0,1,0) \to (0,0,2,0)$$

$$Pr((0,0,2,0)4R \to (0,0,2,0)) \ (\frac{6}{16})^4 = 0.08$$

2 实验过程

2.1 问题 1

题目描述：编程实现 4 轮的 CipherFour 算法 (注意最后一轮有 P 置换)

题解：

python 实现：

```
#plaintext16进制转2进制
plaintext='{:016b}'.format(int(plaintext,16))        # 还是字符串
# k长16*(4+1)r=80bit
k='{:080b}'.format(int(k,16))

```

输入明文 *plaintext* 为 4 位十六进制字符串，密钥 k 为 $4 \cdot (4+1) = 20$ 位 16 进制字符串，k_0 为其中 [0:4] 位，k_0 为其中 [4:8] 位，依次类推。

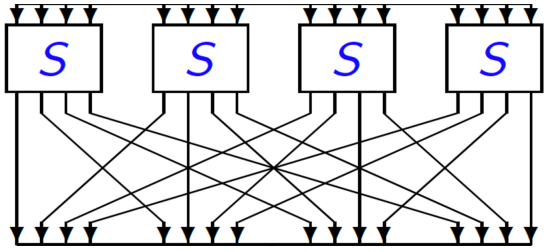
加密前先将明文和密钥转换为二进制，明文与 k_i 均为 16 位

CipherFour 中每一轮加密都经过相同步骤, 在第 i 轮中

1、明文 *plaintext* 与密钥 k_i 逐 bit 异或

2、将异或后的 16 位字符串每四个划分为一组作为 S 盒的输入，代入 S 盒得到 $4 \cdot 4 = 16$ 位输出

3、将经过 S 盒后的 $4 \cdot 4$ 位输出进行 P 置换，每个 S 盒变换后的输出第 1 位依次作为 P 置换结果的 [0:4] 位，变换后的输出第 2 位依次作为 P 置换结果的 [4:8] 位... 依次类推。



经历四次上述步骤后再将明文 *plaintext* 与最后的密钥 k_4 异或，得到密文。

当明文为 0x0020，密钥为 0x5b92064b1e03a55fecbd 时，加密结果为 **0xc13e**。

C++ 实现：

C++ 中我们将 16bit 明密文声明为一个 unsigned short 两字节的整型数据，然后利用移位运算可以快速实现 S 盒和 P 置换，然后就可以实现 Cipherfour 算法。这样是最大限度的减小了空间和时间，提高了速度，这在后续任务中有所体现。

关键代码：

```
u_short S_box[16] = {0x6, 0x4, 0xc, 0x5, 0x0, 0x7, 0x2, 0xe, 0x1, 0xf, 0x3, 0xd, 0x8, 0xa, 0x9, 0xb};
u_short P_box[16] = {0x0, 0x4, 0x8, 0xc, 0x1, 0x5, 0x9, 0xd, 0x2, 0x6, 0xa, 0xe, 0x3, 0x7, 0xb, 0xf};

//输入是16bit的int，输出也是16bit的int
u_short S(u_short in)
{
    u_short out = 0;
    u_short mask[4] = {0x000f, 0x00f0, 0x0f00, 0xf000}; //使用mask的方式进行4bit分组
    for (int i = 0; i < 4; i++)
    {
        out += S_box[(in & mask[i]) >> (4 * i)] << (4 * i); //过S盒
    }
    return out;
}
//输入是16bit的int，输出也是16bit的int
u_short P(u_short in)
{
    u_short out = 0;
    for (int i = 0; i < 16; i++)
    {
        auto tmp = P_box[i] - i;
        if (tmp >= 0)
            out += (in & (0x8000 >> P_box[i])) << tmp;
          //如果P_box[i]-i>0，就需要左移绝对值位
        else
            out += (in & (0x8000 >> P_box[i])) >> (-1 * tmp);
          //如果P_box[i]-i<0，就需要右移绝对值位
    }
    return out;
}
u_short *KeyGen(int R)
{
    R += 1; //R轮加密需要R+1个密钥
    u_short *Key = new u_short[R];
    for (int i = 0; i <= R; i++)
    {
        Key[i] = rand() & 0xffff;
```

```
    }
    return Key;
}
//R轮的Cipherfour算法
u_short cipherfour(u_short m, int R, u_short *K = NULL)
{
    if (K == NULL)        //如果给的是NULL，那就随机生成一组秘钥
        u_short *K = KeyGen(R);
    u_short c = m;
    for (int i = 0; i < R; i++)
    {
        c = P(S(c ^ K[i]));
    }
    c ^= K[R]; //异或最后一个密钥
    return c;
}

```

2.2 问题 2

题目描述：本题要求我们编写实验统计当输入差分固定时，输出差分出现的频率。

题解：

python 实现：

设定输入差分，首先设定为“00f0 “。

通过字典方法建立起计数器，包括从 0000 到 ffff 共 2^{16} 个键值对，每个键为输出差分，值为对应的个数（初始为 0）。

遍历 20 种随机密钥的情况，对于每个密钥，将遍历 0000 到 ffff 共 2^{16} 个输入，并计算相对应固定差分的另一个输入 $0000 \oplus 00f0$ 到 $ffff \oplus 00f0$ 共 2^{16} 个。计算每对输入经过 CipherFour 算法后输出的密文的差分，并在字典中的相应的键值加一。每种密钥下有 2^{16} 个输出差分，遍历 20 个随机密钥后最终的输出差分共有 1310720 个。

我们将计数器字典根据值的数值（也就是根据每个差分出现次数）从高到低进行排序后输出到 csv 文件中。

关键代码如下：

```
def frequency():
    #随机获得一组密钥:
    XOR_in='00f0'
    #计数器:
    # xorlist=['{:016b}'.format(int(str(m),10)) for m in range(2*16)]
    xorlist=[m for m in range(2**16)]
    all_XOR=dict.fromkeys(xorlist,0)

    for gr_key in range(20):#选取20组密钥:
        key=''.join(str(hex(random.randint(0,15))[2:])for i in range(16))#随机key
        for cou_in  in range(2**16):#2**16
```

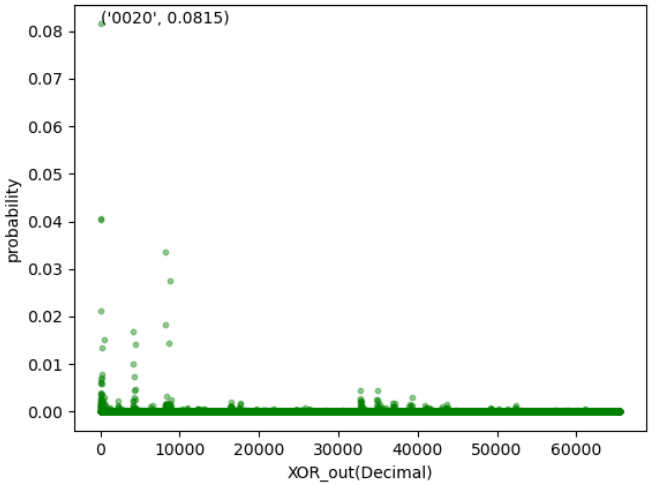
```
            plain1='{:04x}'.format(cou_in)
            ciphertext1 = enc(plain1,key)#密文十进制
            plain2='{:04x}'.format(int(plain1, 16) ^ int(XOR_in, 16))
            ciphertext2 = enc(plain2,key)#密文十进制
            XOR_out=ciphertext1 ^ ciphertext2
            all_XOR[XOR_out]+=1
    sortxor=dict(sorted(all_XOR.items(),key = lambda d:d[1],reverse=True))#根据值排序

    dff=pd.DataFrame(list(sortxor.items()))
    dff.to_csv('frequency_2.2.1.csv',sep=',',index=False,encoding="utf_8_sig")#写入文件

```

由此我们可以得到：

输入异或为 00f0 对应最大输出异或为 00c0，计数器计数 6952，频率约为 0.1060791

输入异或为 0020 对应最大输出异或为 00c0，计数器计数 106862 频率约为 0.081529236

差分 0020 作图如下：



由上图及输出表格内容可以看到各个计数器数值分布，验证了经过差分器后输出差分是不均匀的。计数器计数分布从 0 到 106862 不等。其中，频率最高的为输出差分‘0020’，频率为 0.082, 其次是‘0002’，频率为 0.0406. 共有约 32404 项输出差分频率为 0。

C++ 实现：

和 python 一样，只需要遍历 65536 对 x，统计 $cipherfour(x) \oplus cipherfour(x \oplus 00f0)$ 和 $cipherfour(x) \oplus cipherfour(x \oplus 0020)$ 的数量就可以了。

```
//遍历x in {0,1}^16，统计 cipherfour(x)^cipherfour(x^0x00f0)
void task2(int Keyset_num = 20)
{
    printf("————————————————————task2————————————————————\n");
    int R = 4;
```

```
    u_short **Key = new u_short *[Keyset_num]; //使用动态二维数组
    for (int i = 0; i < Keyset_num; i++)
    {
        Key[i] = new u_short[5];
    }
    KeyGen(R, Keyset_num, Key);//随机生成秘钥
    //建立两张20*out的表，一张是IN=00f0的一张是IN=0020的，把20组密钥的结果加在一起，
    int table_00f0[65536] = {}; //这里不能用u_short数组，会溢出！
    int table_0020[65536] = {};
    clock_t start = clock();
    int inxor_0020_num = 0;
    int inxor_00f0_num = 0;
    for (int i = 0; i < Keyset_num; i++)
    {
        for (int x = 0; x < 65536; x++)
        {
            u_short tmp = CF(x, R, Key[i]);
            table_0020[tmp ^ CF(x ^ 0x0020, R, Key[i])] += 1;
            table_00f0[tmp ^ CF(x ^ 0x00f0, R, Key[i])] += 1;
        }
    }
    //  写入文件，可以用python读文件，画一个分布图
    // ofstream fp1("task2_00f0.txt");
    // ofstream fp2("task2_0020.txt");
    // if(fp1.is_open()&&fp2.is_open())
    // {
    //     for(int i=0;i< 65536;i++)
    //         fp1<<table_00f0[i]<<endl;
    //     for(int i=0;i< 65536;i++)
    //         fp2<<table_0020[i]<<endl;
    // }
    int maxout_00f0 = max_element(table_00f0, table_00f0 + 65536) - table_00f0; //返回的是最大值所在的地址，减去起始地址，就是坐标
    printf("%d组随机秘钥，4轮CF，输入异或:00f0，最大输出异或：%-*x，总数：%d\n", Keyset_num, 4, maxout_00f0, table_00f0[maxout_00f0]);

    int maxout_0020 = max_element(table_0020, table_0020 + 65536) - table_0020;
    printf("%d组随机秘钥，4轮CF，输入异或:0020，最大输出异或：%-*x，总数：%d\n", Keyset_num, 4, maxout_0020, table_0020[maxout_0020]);

    //根据任务三的要求，需要找到 0x0020 ——> 0x0020的概率
    double rate = table_0020[0x0020] / (65536.0 * Keyset_num);
    printf("%d组随机秘钥，4轮CF，输入异或:0020，输出异或：0020，总数：%d，概率：%f\n", Keyset_num, table_0020[0x0020], rate);
    printf("task2共花费%fs\n", ((double)clock() - (double)start) / CLOCKS_PER_SEC);
}

```

运行结果：

如下图，在-O2 优化下，时间可以控制在 0.5s 以内

```
zzp@ubuntu:/mnt/d/junior_term1/密码分析学/差分分析$ make
g++ Cipherfour.cpp -o Cipherfour
./Cipherfour
————————————————————task2————————————————————
20组随机秘钥，4轮CF，输入异或:00f0，最大输出异或：c0  ，总数：5732
20组随机秘钥，4轮CF，输入异或:0020，最大输出异或：20  ，总数：103376
20组随机秘钥，4轮CF，输入异或:0020，输出异或：0020，总数：103376，概率：0.078870
task2共花费0.984375s

zzp@ubuntu:/mnt/d/junior_term1/密码分析学/差分分析$ make
g++ Cipherfour.cpp -o Cipherfour -O2
./Cipherfour
————————————————————task2————————————————————
20组随机秘钥，4轮CF，输入异或:00f0，最大输出异或：c0  ，总数：7010
20组随机秘钥，4轮CF，输入异或:0020，最大输出异或：20  ，总数：104408
20组随机秘钥，4轮CF，输入异或:0020，输出异或：0020，总数：104408，概率：0.079657
task2共花费0.375000s

```

下面将 python 与 c 语言编译执行进行时间对比：

python	c 语言
94.3541606s	0.32s

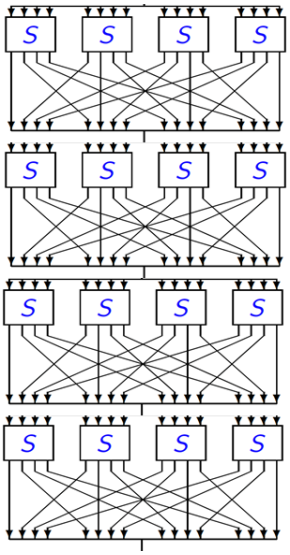
可以看出 C 语言时间效率更高。

2.3 问题 3

题目描述：尝试根据算法及 S 盒差分分布表计算差分 $(0,0,2,0) \xrightarrow{4R} (0,0,2,0)$ 的概率并与 2）的实验结果进行比较。

题解：

由于差分的输入输出不受密钥异或的影响，故四轮 CipherFour 可简化为



第一轮的输入差分为 (0,0,2,0),根据 S 盒差分分布表,输出差分可能为 (0,0,1,0)、(0,0,2,0)、(0,0,9,0)、(0,0,a,0)，经过 P 置换后为 (0,0,0,2)、(0,0,2,0)、(2,0,0,2)、(2,0,2,0) 这四种情况，概率分别为 $\frac{6}{16},\frac{6}{16},\frac{2}{16},\frac{2}{16}$。

当第一轮输出为 (0,0,0,2) 时，将 (0,0,0,2) 作为第二轮的输入差分，经过 S 盒和 P 置换后有（0,0,0,1）、(0,0,1,0)、(1,0,0,1)、(1,0,1,0) 四种情况。当遍历第一轮的四种可能情况后，第二轮的输出结果如下所示：

0001	0.140625	0009	0.017578125	8009	0.005859375	000a	0.017578125	800a	0.005859375
0010	0.140625	0018	0.017578125	8018	0.005859375	0028	0.017578125	8028	0.005859375
1001	0.046875	1009	0.005859375	9009	0.001953125	200a	0.005859375	a00a	0.001953125
1010	0.046875	1018	0.005859375	9018	0.001953125	2028	0.005859375	a028	0.001953125
0002	0.140625	0081	0.017578125	8081	0.005859375	0082	0.017578125	8082	0.005859375
0020	0.140625	0090	0.017578125	8090	0.005859375	00a0	0.017578125	80a0	0.005859375
2002	0.046875	1081	0.005859375	9081	0.001953125	2082	0.005859375	a082	0.001953125
2020	0.046875	1090	0.005859375	9090	0.001953125	20a0	0.005859375	a0a0	0.001953125

在第三轮中和第四轮中重复第二轮的步骤，将上述表格中的所有可能的差分进行遍历、累加，即可得到最终 0020 $\xrightarrow{4R}$ 0020 的概率为 0.0791015625，我们发现这与第二题的概率非常接近，马后炮一下，经过第 5 题的分析，我们知道 0020 $\xrightarrow{4R}$ 0020 的最优差分有 4 条路径，每条都是 $(6/16)^4$，总的概率也是 $4*(6/16)^4 = 0.0791015625$，说明 0020 $\xrightarrow{4R}$ 0020 总共只有 4 条路径，而且每条都是高概率的。

2.4 问题 4

题目描述：利用现成的随机数生成算法，生成 216 对 16 比特的随机数（相当于随机置换的输出对），统计异或值出现的频率并与 2）中结果进行比较。

题解：

本题需要我们检验差分分布表的不均匀性，即对比随机生成的输出差分频率与通过 CipherFour 生成的输出差分频率。

理论分析：对于任意两个四位十六进制数输出差分，考虑任意四位十六进制的均匀分布的概率，应为 $\frac{1}{(16)^4} \approx 0.000015259$.

python 实现：

下面通过代码利用随机数生成算法生成 2^{16} 对十六比特随机数来进行检验。

```
def rand_XOR():
    xorlist=[m for m in range(2**16)]
    all_XOR=dict.fromkeys(xorlist,0)

    for i in range(2**16):
        rand1=random.randint(0,2**16 - 1)
        rand2=random.randint(0,2**16 - 1)
        xor_out=rand1^rand2
        all_XOR[xor_out]+=1
    dff=pd.DataFrame(list(all_XOR.items()))
    dff.to_csv('frequency_2.4.csv',sep=',',index=False,encoding="utf_8_sig")

```

由于输出存在随机性，在两次运行后，输出差分为 0020 对应的计数器数值分别为 0、2，频率分别为 0 和 0.000030517. 实验值在一定程度上符合理论值。**C++ 实现：**

```
//随机生成2^16对16比特随机数，得到他们的差分关系
void task4()
{
    clock_t start = clock();
    printf("\n————————————————————task4————————————————————\n");
    u_short randxor_table[65536] = {};
    for (int i = 0; i < 65536; i++)
    {
        randxor_table[rand() & 0xffff ^ rand() & 0xffff] += 1;
    }
    printf("65536对随机数，异或为0x0020，概率：%f\n", randxor_table[0x0020] / 65536.0);
    ofstream out("task4.txt");
    if (out.is_open())
    {
        out.clear();
        for (int i = 0; i < 65536; i++)
            out << randxor_table[i] << endl;
        out.close();
    }
    printf("task4共花费%fs\n", ((double)clock() - (double)start) / CLOCKS_PER_SEC);
```

```
21  }
22
```

输出结果：

```
1  ————————————————————task4————————————————————
2  65536对随机数，异或为0x0020，概率：0.000015
3  task4共花费0.578125s
4
```

将 python 与 c 语言编译执行进行时间对比：

python	c 语言
0.6098144s	0.5740000s

2.5 问题 5

题目描述：

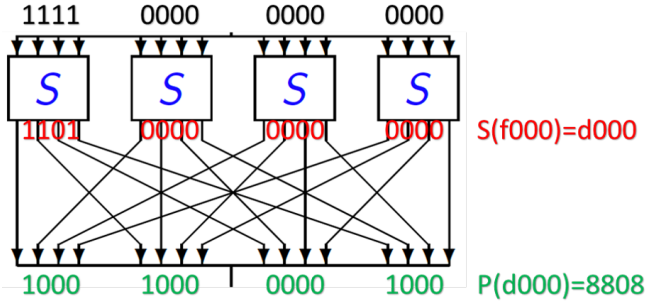
能否找到 4 轮 CipherFour 算法的最优差分特征和最优差分？

2.5.1 题解

要分析 4 轮的 CipherFour 算法的最优差分，我们可以先把问题简化，考虑两轮的 CipherFour 算法。根据 S 盒的差分分布表，4bit 的最优差分是 $P(f \xrightarrow{S} d) = \frac{10}{16}$，但是对于 16bit 的 CipherFour 算法，过完 S 盒还需要过 P 置换，P 置换带来的比特扩散会导致一轮最优差分的后续表现变得很差。

1. 比特扩散导致差分概率下降：

一轮最优差分是 $P(f \xrightarrow{S} d) = \frac{10}{16}$，我们就尝试利用这条特征构造差分路径，比如我们选择输入异或 $\Delta in = f000$，经过 S 盒和 P 置换，就会有如下结果：

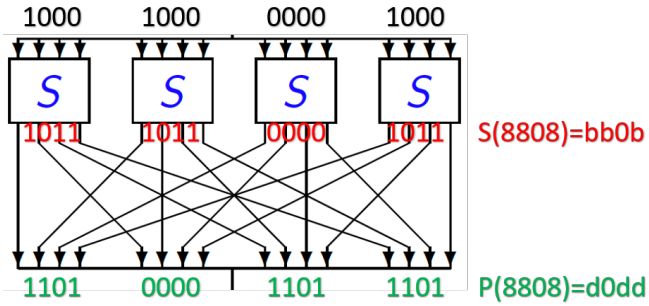


$$Pr(f000 \xrightarrow{S\ P} 8808) = \frac{10}{16}$$

分析第二轮差分的，根据 S 盒的异或分布表，求 8808 分别经过 4 个 S 盒的最大差分概率:

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	6	-	-	-	-	2	-	2	-	-	2	-	4	-
2	-	6	6	-	-	-	-	-	-	2	2	-	-	-	-	-
3	-	-	-	6	-	2	-	-	2	-	-	-	4	-	2	-
4	-	-	-	2	-	2	4	-	-	2	2	2	-	-	2	-
5	-	2	2	-	4	-	-	4	2	-	-	2	-	-	-	-
6	-	-	2	-	4	-	-	2	2	-	2	2	2	-	-	-
7	-	-	-	-	-	4	4	-	2	2	2	2	-	-	-	-
8	-	-	-	-	-	2	-	2	4	-	-	4	-	2	-	2
9	-	2	-	-	-	2	2	2	-	4	2	-	-	-	-	2
a	-	-	-	-	2	2	-	-	-	4	4	-	2	2	-	-
b	-	-	-	2	2	-	2	2	2	-	-	4	-	-	2	-
c	-	4	-	2	-	2	-	-	2	-	-	-	-	-	6	-
d	-	-	-	-	-	-	2	2	-	-	-	-	6	2	-	4
e	-	2	-	4	2	-	-	-	-	-	2	-	-	-	-	6
f	-	-	-	-	2	-	2	-	-	-	-	-	-	10	-	2

因为 $Pr(8 \xrightarrow{S} 8) = Pr(8 \xrightarrow{S} b)$ 不妨取 $8 \xrightarrow{S} b$，就有：



$$Pr(8808 \xrightarrow{S\ P} bb0b) = (\frac{4}{16})^3$$

两轮综合考虑，取输入差分 f000，两轮最高差分概率：

$$Pr(f000 \xrightarrow{2R} d0dd) = \frac{10}{16} * (\frac{4}{16})^3$$

可以看到，因为第一轮中输出异或 d 的比特扩散，导致在第二轮需要同时考虑三个 S 盒的差分概率，也导致后续几轮中需要考虑多个 S 盒的差分，这会极大的降低总的差分路径概率。因此我们要尽量控制比特扩散。比特扩散的关键在于 P 置换，P 置换的本质是一个矩阵的转置，非对角线位置的比特会被扩散到其他 S 盒上：

$$P(\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}) = \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix}$$

因此我们可以选择**输出差分**为 $(1000,0,0,0),(0,0100,0,0),(0,0,0010,0),(0,0,0,0001)$，观察 S 盒的差分分布表，找到高概率的输入差分。如下图，圈出的都是比较高概率的、满足以上四种输出差分的输入差分，注意要对应好 S 盒，比如输出差分选择 $(0,0,0,0001)$，那么输入差分就应该是 $(0,0,0,2)$。

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	6	-	-	-	-	2	-	2	-	-	2	-	4	-
2	-	6	6	-	-	-	-	-	-	2	2	-	-	-	-	-
3	-	-	-	6	-	2	-	-	2	-	-	-	4	-	2	-
4	-	-	-	2	-	2	4	-	-	2	2	2	-	-	2	-
5	-	2	2	-	4	-	-	4	2	-	-	2	-	-	-	-
6	-	-	2	-	4	-	-	2	2	-	2	2	2	-	-	-
7	-	-	-	-	-	4	4	-	2	2	2	2	-	-	-	-
8	-	-	-	-	-	2	-	2	4	-	-	4	-	2	-	2
9	-	2	-	-	-	2	2	2	-	4	2	-	-	-	-	2
a	-	-	-	-	2	2	-	-	-	4	4	-	2	2	-	-
b	-	-	-	2	2	-	2	2	2	-	-	4	-	-	2	-
c	-	4	-	2	-	2	-	-	2	-	-	-	-	-	6	-
d	-	-	-	-	-	-	2	2	-	-	-	-	6	2	-	4
e	-	2	-	4	2	-	-	-	-	-	2	-	-	-	-	6
f	-	-	-	-	2	-	2	-	-	-	-	-	-	10	-	2

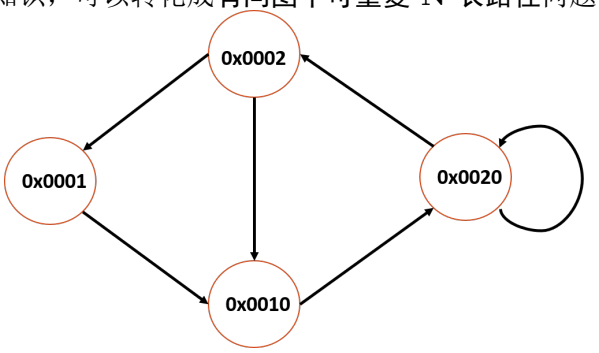
2. 构造 N 轮加密最优差分路径

考虑一轮加密的差分 $\Delta in \xrightarrow{S\ P} \Delta out$，因为要控制 P 置换带来的比特扩散，同时寻找高概率差分，我们可以得到**一轮加密无比特扩散最优差分路径：**

$$Pr(0001 \xrightarrow{S} 0002 \xrightarrow{P} 0010) = \frac{6}{16}Pr(0002 \xrightarrow{S} 0001 \xrightarrow{P} 0001) = \frac{6}{16}Pr(0002 \xrightarrow{S} 0002 \xrightarrow{P} 0010) = \frac{6}{16}Pr(0010 \xrightarrow{S} 00$$

这六种差分路径就是在控制比特扩散下最优的六条路径，在这种情况下，可以直接将输出差分作为第二轮加密的输入差分，而不用考虑扩散。现在问题就变的简单了，只需要利用组合学的知识，匹配输入输出差分，就可以构造出 **N 轮加密的所有最优差分路径**，无论有多少轮，最优差分路径都是以上六条一轮差分路径的组合。

经过分析，我们把将问题转化成了如何使用六条有向路径构建 N 长的路径问题。通过数据结构和离散数学知识，可以转化成**有向图中可重复 N 长路径**问题。



在上述有向图中，所有的可重复（N+1）长路径，就是 N 轮加密的所有最优差分路径，且概率均为 $\frac{6}{16}$，**最优差分特征**是所有最优差分路径中，出现次数最多的、具有相同输入差分和输出差分的所有路径。

有向图的 N 长可重复路径，可以利用图深度优先遍历 DFS 的思想，将节点分为未访问集合和已访问集合，然后利用栈的数据结构来解决。

python 代码实现：

```
# 构造R轮差分的最优差分路径
def task5(R=4):
    # 一轮差分有向图的邻接矩阵，下标0,1,2,3分别对应0001,0002,0010,0020,行表示行标的出度，列表示列表的入度，
    mymap = {0:"0x0001", 1:"0x0002", 2:"0x0010", 3:"0x0020"}
    Adjacency_Matrix = [[0, 0, 1, 0],
                        [1, 0, 1, 0],
                        [0, 0, 0, 1],
                        [0, 1, 0, 1]]
    visited = [[]for i in range(R+1)] # 已访问元素列表，对5个位置都要记录已访问列表
    BDP = []    # 最优差分路径 Best Diff Path
    flag = 1
    for i in range(4):
        stack = []  # 栈，用于记录当前路径的元素信息
        stack.append(i)
        j=0
        flag = 1
        while flag:
            if Adjacency_Matrix[stack[-1]][j]==1 and j not in visited[len(stack)]: # 有向图需要更改遍历的行
                stack.append(j)
                j=0
            else:
                j+=1
            if len(stack)==R+1:
                BDP.append(stack[:])                # 路径长度满足要求，添加到BDP中，
                visited[-1].append(stack.pop()) # 长度为5，所以要把栈顶元素加入到下标4的visited列表中
            elif j==4:  # 到达行的末尾，说明当前栈不可能继续增长，需要回退
                visited[len(stack)].clear() # 先把visited[len(stack)]列表清空，
                tmp = stack.pop()           # 然后弹出栈顶元素，
                visited[len(stack)].append(tmp) # 再把栈顶元素加入到
                j=0     # 从头遍历
            if not stack:
                flag=0
    print("----------------最优差分路径----------------")
    print("\n{}轮最优差分路径共{}条，概率均为{}".format(R,len(BDP),pow(6./16,R)))
    for path in BDP:
        for item in path:
            print(mymap[item],end=" -> ")
        print("\b\b\b   ")

    print("\n----------------最优差分特征----------------\n")
    for i in range(4):
        rank = [0]*4
        #print("输出差分为{}".format(mymap[i]))
```

```
        for path in BDP:
            if path[-1] == i:
                rank[path[0]]+=1
                # for item in path:
                #     print(mymap[item],end=" -> ")
                # print("\b\b\b    ")
        for j in range(4):
            if rank[j]!=0:
                print("输入差分:{}——>输出差分:{} 共有{}条路径".format(mymap[j],mymap[i],
    rank[j]))
        print()
if __name__=="__main__":
    task5()
```

执行结果：cmd 打印结果

```
PS D:\junior_term1\密码分析学\差分分析> python task5.py
----------------最优差分路径----------------

4轮最优差分路径共25条，概率均为0.019775390625
0x0001 -> 0x0010 -> 0x0020 -> 0x0002 -> 0x0001
0x0001 -> 0x0010 -> 0x0020 -> 0x0002 -> 0x0010
0x0001 -> 0x0010 -> 0x0020 -> 0x0020 -> 0x0002
0x0001 -> 0x0010 -> 0x0020 -> 0x0020 -> 0x0020
0x0002 -> 0x0001 -> 0x0010 -> 0x0020 -> 0x0002
0x0002 -> 0x0001 -> 0x0010 -> 0x0020 -> 0x0020
0x0002 -> 0x0010 -> 0x0020 -> 0x0002 -> 0x0001
0x0002 -> 0x0010 -> 0x0020 -> 0x0002 -> 0x0010
0x0002 -> 0x0010 -> 0x0020 -> 0x0020 -> 0x0002
0x0002 -> 0x0010 -> 0x0020 -> 0x0020 -> 0x0020
0x0010 -> 0x0020 -> 0x0002 -> 0x0001 -> 0x0010
0x0010 -> 0x0020 -> 0x0002 -> 0x0010 -> 0x0020
0x0010 -> 0x0020 -> 0x0020 -> 0x0002 -> 0x0001
0x0010 -> 0x0020 -> 0x0020 -> 0x0002 -> 0x0010
0x0010 -> 0x0020 -> 0x0020 -> 0x0020 -> 0x0002
0x0010 -> 0x0020 -> 0x0020 -> 0x0020 -> 0x0020
0x0020 -> 0x0002 -> 0x0001 -> 0x0010 -> 0x0020
0x0020 -> 0x0002 -> 0x0010 -> 0x0020 -> 0x0002
0x0020 -> 0x0002 -> 0x0010 -> 0x0020 -> 0x0020
0x0020 -> 0x0020 -> 0x0002 -> 0x0001 -> 0x0010
0x0020 -> 0x0020 -> 0x0002 -> 0x0010 -> 0x0020
0x0020 -> 0x0020 -> 0x0020 -> 0x0002 -> 0x0001
0x0020 -> 0x0020 -> 0x0020 -> 0x0002 -> 0x0010
0x0020 -> 0x0020 -> 0x0020 -> 0x0020 -> 0x0002
0x0020 -> 0x0020 -> 0x0020 -> 0x0020 -> 0x0020

----------------最优差分特征----------------

输入差分:0x0001——>输出差分:0x0001 共有1条路径
输入差分:0x0002——>输出差分:0x0001 共有1条路径
输入差分:0x0010——>输出差分:0x0001 共有1条路径
```

```
输入差分:0x0020——>输出差分:0x0001 共有1条路径

输入差分:0x0001——>输出差分:0x0002 共有1条路径
输入差分:0x0002——>输出差分:0x0002 共有2条路径
输入差分:0x0010——>输出差分:0x0002 共有1条路径
输入差分:0x0020——>输出差分:0x0002 共有2条路径

输入差分:0x0001——>输出差分:0x0010 共有1条路径
输入差分:0x0002——>输出差分:0x0010 共有1条路径
输入差分:0x0010——>输出差分:0x0010 共有2条路径
输入差分:0x0020——>输出差分:0x0010 共有2条路径

输入差分:0x0001——>输出差分:0x0020 共有1条路径
输入差分:0x0002——>输出差分:0x0020 共有2条路径
输入差分:0x0010——>输出差分:0x0020 共有2条路径
输入差分:0x0020——>输出差分:0x0020 共有4条路径
```

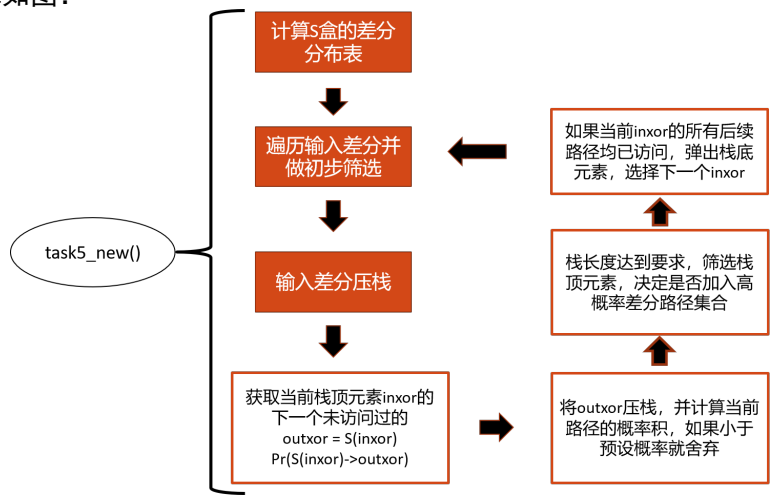
2.5.2 拓展

寻找任意 S 盒 N 轮加密的高概率差分路径和差分特征

4 轮最优差分特征只有最后两个 S 盒是非零比特，对于前两个 S 盒的破解没有帮助，如果我们能找到前两个 S 盒的高概率差分特征，就可以通过 task6，解出 K5 的所有比特。

类比最优差分的构造，任意差分的构造也是利用有向图的深度优先遍历实现的，但是这是一个复杂的多的代码工程。在最优差分的构造中，我们的有向图，只包括最高概率的无比特扩散的一轮差分路径，而在任意差分路径的构造过程中，需要考虑 S 盒差分分布表上所有的非零位置。也就是说，这是一个加权有向图，而这个**有向图的邻接矩阵就是 S 盒的差分分布表**。

代码逻辑如图：



相比 task5()，task5_new() 困难在于每个 inxor 去寻找 S(inxor) 的过程都需要分成 4

组，经过 4 个 S 盒，这也让已访问集合（Visited Set）变成一个高维的列表，同时 Visited Set 需要随着栈顶元素的更新而更新，也让我深深体会到了**数据结构与算法的重要性**，和 debug 带来的折磨。也能感觉出，差分可能并不是一个足够优秀的方法，只是一种变相的穷举的剪枝，不过如果能在数学上提供高概率的差分条件，那么穷尽算法，就会像 task5() 一样变成简单的 DFS 算法，并且**加密轮数的增加只会降低差分路径的概率而不会改变差分特征**。也就意味着该密码算法的破解难度直线下降（时间和空间上）。

python 实现，关键函数：

因为数据结构过于复杂，使用 python 可以降低不少工作量，但其实也少多少。。。

```
def S_diff(inxor,stack_len):
    ''' 函数通过传入的inxor和S盒的差分分布表（邻接矩阵），寻找下一个未访问过的S(inxor)并返回其概率 '''
    pr = 1
    flag = 1
    loc = [0,0,0,0]      # 记录遍历的起始点，没有也可以
    while flag:
        re_i = 0
        reS_out = 0      # 设置了很多标志，因为有多层循环，需要各种控制标志来决定程序的执行流程
        out = 0          # S(inxor)是一个16bit的整数
        pr=1
        for i in range(4): # 有四个S盒，从最右边的S盒开始遍历，注意这里的编号，最右的S盒记为S_0
            if re_i: # 当第i个S盒的所有S_out都已经访问过，就要先寻找第i+1个S盒的下一个元素，再返回到第0个S盒重新遍历
                break
            S_in = (inxor&Mask[i])>>(4*i)   # 取出第i个S盒对应的4bit作为S_in
            for S_out in range(loc[i],16): # 遍历邻接矩阵，S_out
                tmp = S_table[S_in][S_out] # 取出权重(概率)
                #print(Visited[stack_len][i])
                if tmp > 0 and S_out not in Visited[stack_len][i]: # 如果概率大于0，并且没有访问过
                    if reS_out:
                        '''如果设置了reS_out标志，说明第i-1个S盒已经全部访问过了，所以现在是在找第i个S盒的下一个元素，找到之后就把这个元素加入到已访问列表中，'''
                        Visited[stack_len][i].append(S_out)
                        loc[i] += 1
                        reS_out = 0
                        re_i = 1    # 找到了未访问的第i个S盒的元素，然后重新从最右的S盒开始遍历
                        break
                    out += S_out << (4 * i) # 结果out是四个S盒的输出的级联
                    pr *= tmp       # 概率是四个S盒的概率积
                    if i==3:        # 如果已经遍历到最左的S盒了，就可
                        return out, pr
                    break
                elif S_out==15:      # 如果遍历到最0xf还没有满足条件的，说明都已经访问过了
```

```
32                      for k in range(i+1):    # 重置右边S盒的已访问集合和位置信息集合
33                          loc[k] = 0
34                          Visited[stack_len][k].clear()
35                      if i==3:        # 如果是最左的盒子的最后一个元素，说明当前inxor所有可能的S盒的输出都被访问过了
36                          return False,0      # 返回false
37                      reS_out = 1             # 设置信号reS_out
38                      continue
39
40  '''清空已访问集合的部分位置'''
41  def clear_visited(begin):
42      for i in range(begin+1,5):
43          Visited[i]=[[]for i in range(4)]
44
45  # 通过S盒线性分布表，利用DFS，构造N轮加密，一定条件下的所有高概率差分路径并找到高概率差分特征
46  def task5_new(S_num):
47      GDP = []    # good differ path 好的差分路径，hhh，再叫best就不好了
48      '''计算S差分分布表,并将每个位置都除16,就是一张加权的有向图'''
49      for i in range(16):
50          for j in range(16):
51              inxor = i ^ j
52              outxor = S(i) ^ S(j)
53              S_table[inxor][outxor] += 1
54      for i in range(16):
55          for j in range(16):
56              S_table[i][j] /= 16. # 都除16，得到概率就是一个加权有向图的邻接矩阵了
57      #testset = [0x0002,0x0001,0x0020,0x0010] # 用于固定要查找的输入异或
58      testset = [0x0500]
59      for inxor in range(1,65536):            # 遍历输入差分
60          str_xor = "{:0>4}".format(hex(inxor)[2:])
61          if str_xor.count('0')<2:            # 输入差分至少有两个S盒为0，剪枝，去掉低概率路径
62              continue
63          if str_xor[S_num]=='0':             # 指定S盒，确保第S_num个S盒位置的比特非零,从左到右编号为0-3
64              continue
65          # if inxor not in testset:          # 通过固定输入差分到testset测试代码正确性
66          #     continue
67          stack = []                          # DFS深度优先遍历用到的栈结构
68          stack.append(inxor)
69          flag = 1                            # 当栈为空，flag置0，压入下一个输入异或的路径
70          pr_stack = [1]                      # 计算概率的乘积，在栈底留一个元素，防止空列表pop错误
71          while flag:                         # 开始寻找 inxor对应的所有高概率差分路径
72              pathpr=0                        # 储存当前栈上的路径的概率乘积
73  '''核心函数，stack[-1]是栈顶元素，函数返回下一个outxor = S(stack[-1]),以及 pr = Pr(S(stack[-1])->outxor)'''
74              outxor,pr = S_diff(stack[-1],len(stack))
75              if outxor:  # 如果有outxor输出
```

```
76                  pr_stack.append(pr)          # 概率压栈
77                  pathpr = reduce(lambda x,y:x*y,pr_stack)    # 计算当前路径的概率乘积
78                  if pathpr > Pr:   # 如果当前路径的概率已经小于一个预设值Pr，那么就抛弃当前路径，也是剪枝的思想
79                          stack.append(P(outxor)) # 如果概率比预设Pr大，就压入栈
80                  else:
81                      pr_stack.pop()           # 弹出栈顶元素的概率
82                      tmp = P_inv(stack.pop()) & 0x000f  #加入到栈顶的是P置换之后的元素，需要经过逆置换再加入Visited
83                      pr_stack.pop()           # stack.pop()之后一定要跟一个pr.pop
84              '''更新已访问集合，把弹出的栈顶元素的最后一个位置加入已访问，在S_diff函数中就可以避免重复访问或漏掉元素'''
85                      Visited[len(stack)][0].append(tmp)
86                      clear_visited(len(stack)) # 清空栈高位的已访问集合，因为栈底层的元素变动了，所以栈顶方向的元素需要重新遍历
87              else:
88                  '''如果当前栈顶元素inxor的所有S(inxor)都已经遍历过了，返回值就是False，进入else，栈弹出一个元素，相当于回退'''
89                  tmp = P_inv(stack.pop())&0x000f
90                  pr_stack.pop()
91                  Visited[len(stack)][0].append(tmp)
92                  clear_visited(len(stack))      # 同上述作用
93              if len(stack)==R+1:
94                  '''栈的长度已经达到R轮加密的要求，可以加入BDP，同时将当前栈顶元素的最后4bit加入相应的已访问队列'''
95                  str_outxor = "{:0>4}".format(hex(P(outxor))[2:])
96                  if str_outxor.count('0') > 1 and str_outxor[S_num] != '0':
97                      GDP.append((stack[:],pathpr)) # 这里还要筛选一下输出异或，去除输出异或中第S_num个S盒为0的输出差分
98                  tmp = outxor & 0x000f
99                  Visited[-1][0].append(tmp)  # 更新已访问集合，
100                 stack.pop()
101                 pr_stack.pop()
102             elif len(stack)==0:   # 如果当前输异或inxor的所有路径都已经访问过了，栈就会回退为空，然后我们结束while
103                 flag=0
104     print("\n从左向右第{}个S盒的高概率差分路径共{}条".format(S_num+1,len(GDP)))
105
106     # print("\n----------------最优差分路径----------------\n")
107     # for path in GDP:
108     #     for i in range(5):
109     #         print(hex(path[0][i]),end=" ")
110     #     print(path[-1])
111
112     # 统计差分特征
113     diff_pr = dict()
114     for path in GDP:
115         inxor,outxor,pr_ = hex(path[0][0]),hex(path[0][-1]),path[-1]
116         tmp = diff_pr.get((inxor,outxor))
117         if tmp:
```

```
                diff_pr[(inxor,outxor)] += pr_
            else:
                diff_pr[(inxor,outxor)] = pr_
        d_order = sorted(diff_pr.items(), key=lambda x: x[1],reverse=True)

        print("\n————————————从左向右，第{}个S盒最优的10个差分特征————————————\n".format(S_num+1))
        for i in range(10):
            print(d_order[i])

```

输出结果：（cmd 打印内容）

```
PS D:\junior_term1\密码分析学\差分分析> python task5.py
请输入要寻找的S盒的差分特征（从左到右编号为1—4）：1

从左向右第1个S盒的高概率差分路径共580条

————————————从左向右，第1个S盒最优的10个差分特征————————————

(('0x2000', '0x2000'), 0.02108001708984375)
(('0x1000', '0x1000'), 0.011028289794921875)
(('0x1000', '0x2000'), 0.010540008544921875)
(('0x2000', '0x1000'), 0.010540008544921875)
(('0x2020', '0x2002'), 0.006591796875)
(('0x5000', '0x2000'), 0.00640869140625)
(('0x2002', '0x2002'), 0.0053558349609375)
(('0x2020', '0x2020'), 0.0053558349609375)
(('0x2200', '0x2200'), 0.0045318603515625)

PS D:\junior_term1\密码分析学\差分分析> python task5.py
请输入要寻找的S盒的差分特征（从左到右编号为1—4）：2

从左向右第2个S盒的高概率差分路径共673条

————————————从左向右，第2个S盒最优的10个差分特征————————————

(('0x200', '0x200'), 0.0127716064453125)
(('0x200', '0x2200'), 0.00823974609375)
(('0x202', '0x220'), 0.0070552825927734375)
(('0x220', '0x220'), 0.006591796875)
(('0x100', '0x100'), 0.00638580322265625)
(('0x100', '0x200'), 0.00638580322265625)
(('0x200', '0x100'), 0.00638580322265625)
(('0x220', '0x202'), 0.0061798095703125)
(('0x202', '0x202'), 0.0054073333740234375)
(('0x200', '0x202'), 0.00494384765625)

PS D:\junior_term1\密码分析学\差分分析> python task5.py
请输入要寻找的S盒的差分特征（从左到右编号为1—4）：3

```

```
从左向右第3个S盒的高概率差分路径共2200条

————————————从左向右，第3个S盒最优的10个差分特征————————————

(('0x20', '0x20'), 0.0791015625)
(('0x10', '0x10'), 0.0457305908203125)
(('0x20', '0x10'), 0.04119873046875)
(('0x10', '0x20'), 0.03955078125)
(('0xc0', '0x20'), 0.02801513671875)
(('0x50', '0x20'), 0.0263671875)
(('0x80', '0x80'), 0.02108001708984375)
(('0x30', '0x80'), 0.0186767578125)
(('0x20', '0x2020'), 0.01812744140625)
(('0x60', '0x10'), 0.0164794921875)

PS D:\junior_term1\密码分析学\差分分析> python task5.py
请输入要寻找的S盒的差分特征（从左到右编号为1—4）：4

从左向右第4个S盒的高概率差分路径共2264条

————————————从左向右，第4个S盒最优的10个差分特征————————————

(('0x2', '0x2'), 0.0457305908203125)
(('0x1', '0x1'), 0.0259552001953125)
(('0x2', '0x1'), 0.02368927001953125)
(('0x1', '0x2'), 0.02286529541015625)
(('0x2', '0x2002'), 0.01709747314453125)
(('0xc', '0x2'), 0.0160675048828125)
(('0x5', '0x2'), 0.01483154296875)
(('0x8', '0x8'), 0.011272430419921875)
(('0x3', '0x8'), 0.00958251953125)
(('0x1', '0x1001'), 0.009372711181640625)
PS D:\junior_term1\密码分析学\差分分析>
```

2.6 问题 6

以恢复第 5 轮第 3 个 4bit 密钥为例：

选择 4 轮高概率差分路线，由以上第 5 题我们得出最优的差分路线为 $(0,0,2,0)) \xrightarrow{4r} (0,0,2,0)$，概率 $P \approx 0.08 >> \frac{1}{2^{16}}$，所以可以作为我们的差分区分器。

1. 采样：

随机选取明文对 (m, m') 满足 $m \oplus m' = (0,0,2,0)$，具体的，我们可以随机选取 m，然后计算 $m' = m \oplus (0,0,2,0)$ 这样我们就得到了满足头部差分的明文对。

2. 去噪：

我们选取的最优差分路线为 $(0,0,2,0)) \xrightarrow{4r} (0,0,2,0)$，然后继续分析输入差分为 (0 0 2 0) 的第 5 轮的输出，观察 S 盒的输入差分表，输入差分为 0 时，输出差分也为 0，而输入

差分为 2 时,输出差分可能为 $1, 2, 9, a$,也就是说,输出差分不为 $(0,0,1,0),(0,0,2,0),(0,0,9,0),(0,0,a,0)$ 的对一定是错误对,这样我们只需要将输出差分为 $(0,0,1,0),(0,0,2,0),(0,0,9,0),(0,0,a,0)$ 入输出差分对提取出来，即可完成去噪的过程。

3. 密钥恢复：

当我们使用过滤后的明密文对，遍历密钥 $k_{5,2}$, 的 16 种可能，代入方程 $S^{-1}(k_{5,2}) \oplus c_2) \oplus S^{-1}(k_{5,2} \oplus c_2') = 2$ 后，如果明密文对正确，一定可以得出一个包含正确密钥 $k_{5,2}$ 的解集，而对于错误的对，解集中均为错误的密钥 $k_{5,2}$, 错误的对用正确的密钥代入方程一定得不到 2）。

密钥 $k_{5,2}$, 的每种可能对应一个计数器 $key[k]$，初始值为 0，若 k 满足方程 $S^{-1}(k_{5,2}) \oplus c_2) \oplus S^{-1}(k_{5,2} \oplus c_2') = 2$，则将计数器加 1，选取计数器最大的密钥作为我们要求的结果。

4. 成功率分析：

选择 100 对满足输入差分为 (0,0,2,0) 的明文对，其中正确对的个数为 $100 \times 0.08 = 8$，所以正确密钥被计数 8 次，过滤后明文对总数约为 $100 \times 0.11 = 11$ 对，每个错误密钥平均被计数 100×0.0275=2.75 次，8 与 2.75 差距不大，在计数过程中可能会出现计数最高的为某个错误密钥的情况。

而当选择 500 对时，正确密钥被计数约 40 次，错误密钥被计数约 13 次，这样二者差距很大，出现以上错误的概率几乎为 0。

5. 实验结果：

正确密钥：K52=9

```
PS D:\Cryptanalysis\project\project2> & D:/PY365/python.exe d:/Cryptanalysis/project/project2/5.py
密钥K52为: [9]
```

选择 100 对时成功概率：0.676

选择 200 对时成功概率：0.917

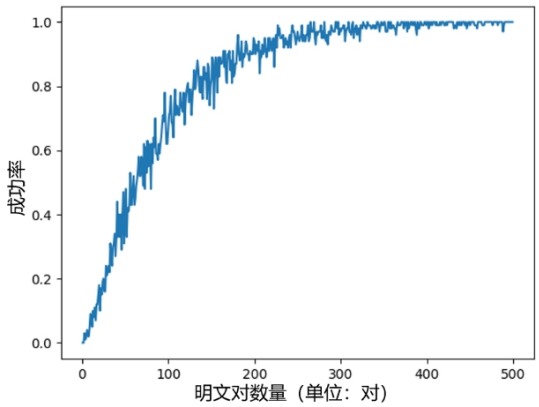
选择 100 对时成功概率：0.973

选择 100 对时成功概率：0.995

选择 500 对时成功概率：0.999

```
选取100个明文对时成功概率为: 0.671
选取200个明文对时成功概率为: 0.917
选取300个明文对时成功概率为: 0.973
选取400个明文对时成功概率为: 0.995
选取500个明文对时成功概率为: 0.999
```

详细的，我们可以通过作图来更加直观的观察出随着明文对数的增加，成功率（得出正确密钥的概率）的变化。



此外我们还求得了其他 3 个密钥：

组合起来即为：0x889e

```
PS D:\Cryptanalysis\project\project2> & D:/PY365/python.exe d:/Cryptanalysis/project/project2/5.py
密钥K50为：[8]
密钥K51为：[8]
密钥K52为：[9]
密钥K53为：[14]
```