Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
Department of Mechanical Engineering

6.050J/2.110J **Information and Entropy** Spring 2003

Problem Set 1 Solutions

Solution to Problem 1: Completely Logical

We are asked to check the boolean equality $(\overline{A \cdot B}) + \overline{B} = \overline{A \cdot B}$. Here we present three ways to test this identity: direct algebraic manipulation, hand writing the truth tables, and a MATLAB calculation.

Algebraic Manipulation

The following manipulations are drawn from Table 1.4, Properties of Boolean Algebra in the course notes.

$$\begin{aligned}
(\overline{A \cdot B}) + \overline{B} &= (\overline{A} + \overline{B}) + \overline{B} && \text{(de Morgan)} \quad (1)\\
&= \overline{A} + (\overline{B} + \overline{B}) && \text{(Associative)} \quad (2)\\
&= \overline{A} + \overline{B} && \text{(Idempotent)} \quad (3)\\
&= \overline{A \cdot B} && \text{(de Morgan)} \quad (4)
\end{aligned}$$

Thus we conclude that the identity is true.

Truth Table Analysis

We can also prove this identity by writing out the truth tables. By hand it looks like this, with all intermediate calculations:

A	B	$\overline{B}$	$A \cdot B$	$\overline{A \cdot B}$	$\overline{A \cdot B} + \overline{B}$
0	0	1	0	1	1
0	1	0	0	1	1
1	0	1	0	1	1
1	1	0	1	0	0

Table 1: Truth table analysis of Problem 1

MATLAB Program

Using the following code, we can solve the problem in MATLAB:

```
% Start of file, filename testequality.m
% Check the equation OR(NAND(A,B), NOT(B)) = NAND(A,B)
%

A = [0 0 1 1];
B = [0 1 0 1];
Right = ~(A & B);
```

```
Left = (Right)|(~B);
if Left == Right
    fprintf('The equality is true.\n');
  else
    fprintf('The equality is false.\n');
end

%%% End of file
```

This code gives the following output:

```
>> testequality
The equality is true.
```

Solution to Problem 2: Universality

This particular definition of universality only treats arbitrary functions of **two** Boolean variables, but with any number of outputs. It appears to be an onerous task to prove universality for an arbitrary number of outputs. However, since each individual output of a multi-output function can be considered a separate one-ouput function, it is sufficient to prove the case of only one-output functions. This is why we begin by listing all one-output functions of one variable.

Solution to Problem 2, part a.

Each variable A and B has two possible values, making four different combinations of inputs (A, B). Each combination of inputs (four possible) can cause one of two output values. Therefore the number of possible one-output binary functions of two binary variables is 2^4, or 16. They are enumerated in the table below.

		$F(A,B)$															
A	B	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Table 2: Truth table analysis of Problem 2, part a

Solution to Problem 2, part b.

OR is not a universal function, and this can be seen most easily by noting that it is monotonic increasing; in other words, that in any nested *OR* function, once you get a 1, you will always have a 1. Thus you could never implement a *NOT* gate with nested *OR* functions: giving a 0 to a *NOT* gate gets you a 1 (an **increase** in the output), but giving a 1 to a *NOT* gets you a 0, which is a **decrease** in the output. Since the output of an *OR* gate is always the same as or greater than its input, a *NOT* gate can never be implemented.

Solution to Problem 2, part c.

AND cannot be a universal function for a similar reason: *AND* is monotonic increasing, and thus a *NOT* gate cannot be constructed.

Solution to Problem 2, part d.

We will present a proof based on parity that XOR is not universal. Consider a finitely nested function of XORs which is ultimately a function of two inputs A and B and any finite number of constants 0 and 1. That is, we define a nested function of XORs to be an expression f drawn from the set EXPR={0, 1, A, B, or, $XOR(f_\alpha, f_\beta)$}, where f_α and f_β are also drawn from the set EXPR, but only a finite number of times. Consider the property E which is the even parity of the four values $f_\gamma(0,0)$, $f_\gamma(0,1)$, $f_\gamma(1,0)$, and $f_\gamma(1,1)$. That is, a function f_γ has property E if and only if 0, 2, or 4 of these values is 1. Thus A has property E, as does B, 0, and 1.

If our function f is equal to $AND(A,B)$, then it does not have property E. Therefore f cannot be equal to A, B, 0, or 1, and so it must be of the form $XOR(f_\alpha, f_\beta)$. An XOR function that has its output without property E has exactly one of its two inputs with property E and one without. This can be seen by noting that XOR produces a 1 only upon input of a 1 and a 0, i.e., an 'unpaired' 1. Thus the number of 1's in the output is equal to the number of unpaired 1's in the input. If there are an odd number of 1's in the output, then the number of unpaired 1's in one input must be even and the number of unpaired ones in the other input must be odd (an odd number can only be a sum of an even number and odd number). Since the number of paired 1's in each input are by definition equal (call this number x), the number of 1's in one of the inputs is $x + e$, and the number of 1's in the other input is $x + o$, where e and o are even and odd numbers, respectively. Thus we have exactly one even and an one odd input. Returning to the main proof, where does the input without property E come from? Only another XOR function. Eventually you run out of functions (because we assumed finite) and thus f cannot be equal to $AND(A,B)$.

Solution to Problem 2, part e.

NAND does not succumb to the type of proof shown above, and in fact it is a universal function. The easiest way to check this is to enumerate all possible cases, and the easiest way to do that is with MATLAB. The following program (with associated subfunction, which exists as a separate file) does the test:

```
% Start of file, filename fnand.m
% Subfunction NAND(A,B)
%

function[C]=fnand(A,B)
C=~(A & B);

%%% End of file
```

```
% Start of file, filename nanduniv.m
% Demonstrate universality of NAND gate
%

A=[0 0 1 1]
B=[0 1 0 1]
N=[1 1 1 1];
Z=[0 0 0 0];

b_15=N         % Ones
b_14=fnand(A,B)% NAND(A,B)
b_13=fnand(b_14,A)
b_12=fnand(A,1)% NOT(A)
```

```
b_11=fnand(b_12,B)
b_10=fnand(B,1)% NOT(B)
b_9=fnand(fnand(A,B),fnand(b_12,b_10))
b_8=fnand(fnand(b_10,b_12),1)
b_7=fnand(b_8,1)
b_6=fnand(b_9,1)
b_5=fnand(b_10,1)
b_4=fnand(b_11,1)
b_3=fnand(b_12,1)
b_2=fnand(b_14,1)
b_1=fnand(b_13,1)
b_0 = Z       % Zeros

%%% End of file
```

This code gives the following output:

```
>> fnanduniv
A =
  0 0 1 1
B =
  0 1 0 1
b_15 =
  1 1 1 1
b_14 =
   11 1 0
b_13 =
  1 1 0 1
b_12 =
  1 1 0 0
b_11 =
  1 0 1 1
b_10 =
  1 0 1 0
b_9 =
  1 0 0 1
b_8 =
  1 0 0 0
b_7 =
  0 1 1 1
b_6 =
  0 1 1 0
b_5 =
  0 1 0 1
b_4 =
  0 1 0 0
b_3 =
  0 0 1 1
b_2 =
  0 0 1 0
b_1 =
  0 0 0 1
b_0 =
  0 0 0 0
```

Solution to Problem 3: Make it Fit

Solution to Problem 3, part a.

There are 96 characters we need to encode: lower case (26), upper case (26), digits (10), space (1), punctuation marks (32) and ETX (1). The Caltech code uses 28 out of a possible 32 5-bit sequences, and uses all of the 9-bit sequences, of which there are 64 (only 6 of the 9 bits are used, giving $2^6 = 64$ codes). Thus the Caltech code has only 92 spots and is not large enough for all of our characters. The MIT code, on the other hand, uses all 32 of its 6-bit sequences, and has 64 spots left in the 7-bit sequences, giving a total of 96 spots in the code, just enough to fit all of the characters desired. Thus the MIT code works and the Caltech code does not, as one would naturally expect.

Solution to Problem 3, part b.

The Caltech graduate's code does not work. One possible redefinition of it is as follows: There are four unused 5-bit codes 11100, 11101, 11110, and 11111. Suppose we reclaim the next lowest 5-bit code, namely 11011. Now we have 27 5-bit codes to assign. We will assign these to the 26 lower case letters and space (the most commonly occurring characters). Then we will add four bits to the unused 5-bit codes (call these headers), giving us another 80 unique codes (4-bits per header gives 16 unique codes per header, times five headers equals 80 unique codes). This gives us a total of 107 unique codes (27 5-bit and 80 9-bit) which is enough to fit all of the symbols we need, plus extras if we wish. Note that this code has become more complicated to decode, since you have to keep watch for five header bits instead of one.

Solution to Problem 3, part c.

The following table compares the MIT code, ASCII, the 8-bit computer code, and the modified Caltech code presented in previous subsolution . Remember that ASCII code uses 7 bits per character no matter what, and 8-bit computer code is just that, 8-bits per character.

		# of bits required			
Char. type	# of Chars	MIT Code	ASCII	8-bit	Modified Caltech Code
lower case	200	1200	1400	1600	1000
space	50	300	350	400	250
upper case	20	140	140	160	180
common punc.	5	30	35	40	45
uncommon punc.	20	140	140	160	180
ETX	1	6	7	8	9
Total	296	1816	2072	2368	1664
Average Bits/Char.	–	6.13	7	8	5.62

Table 3: Comparison of character codes for Problem 3, part c

Solution to Problem 3, part d.

To modify the MIT graduate's code to accomodate 12 extra characters, one possible solution is as follows: The 12 additional characters are relatively rare, and so we can afford to use a longer sequence of 8 bits for them. Additionally, to distinguish them from the other sequences in our code, we could specify that they all start with 111. However, this will invalidate 16 of the 7-bit sequences (namely, all of those which start with 111). So we have newly available all the 8-bit codes that start with 111, or 32 codes, minus the 7-bit codes that start with 111, so we have a total of 16 new 8-bit codes. The extra 12 characters can be distributed among these codes.

Solution to Problem 3, part e.

The following two MATLAB programs implement an encoder and decoder for a select few letters of the MIT graduate's code. The first program is 'encode', and it takes a string of alphanumeric characters and outputs a sequence of 1's and 0's ordered according to the MIT Code into the variable `encodedmessage`. If it encounters a character it does not recognize, it inserts a space.

```
% Start of file, filename encode.m
% MIT Code Encoder
%

%%% Get string to encode
message = input('Input a string you would like encoded
(please use only the first seven letters, space, period,
and exclamation point):','s');

messagelength=length(message);

encodedmessage=[];

%%% Encode each letter and append it to encodedmessage for i=1:messagelength
  switch message(i)
    case {'a'}
      newcharacter=0;
      newcharacter=de2bi(newcharacter,6, 'left-msb');
    case {'b'}
      newcharacter=1;
      newcharacter=de2bi(newcharacter,6, 'left-msb');
    case {'c'}
      newcharacter=2;
      newcharacter=de2bi(newcharacter,6, 'left-msb');
    case {'d'}
      newcharacter=3;
      newcharacter=de2bi(newcharacter,6, 'left-msb');
    case {'e'}
      newcharacter=4;
      newcharacter=de2bi(newcharacter,6, 'left-msb');
    case {'f'}
      newcharacter=5;
      newcharacter=de2bi(newcharacter,6, 'left-msb');
    case {'g'}
      newcharacter=6;
```

```
        newcharacter=de2bi(newcharacter,6, 'left-msb');
      case {' '}
        newcharacter=27;
        newcharacter=de2bi(newcharacter,6, 'left-msb');
      case {'.'}
        newcharacter=28;
        newcharacter=de2bi(newcharacter,6, 'left-msb');
      case {'!'}
        newcharacter=29;
        newcharacter=de2bi(newcharacter,6, 'left-msb');
      case {'A'}
        newcharacter=65;
        newcharacter=de2bi(newcharacter,7, 'left-msb');
      case {'B'}
        newcharacter=66;
        newcharacter=de2bi(newcharacter,7, 'left-msb');
      case {'C'}
        newcharacter=67;
        newcharacter=de2bi(newcharacter,7, 'left-msb');
      case {'D'}
        newcharacter=68;
        newcharacter=de2bi(newcharacter,7, 'left-msb');
      case {'E'}
        newcharacter=69;
        newcharacter=de2bi(newcharacter,7, 'left-msb');
      case {'F'}
        newcharacter=70;
        newcharacter=de2bi(newcharacter,7, 'left-msb');
      case {'G'}
        newcharacter=71;
        newcharacter=de2bi(newcharacter,7, 'left-msb');
      otherwise
        newcharacter=27;
        newcharacter=de2bi(newcharacter,6, 'left-msb');
      end
    encodedmessage=cat(2,encodedmessage,newcharacter);
  end

  %%% Add the ETX character encodedmessage=cat(2,encodedmessage, de2bi(31,6,'left-msb'))

  %%% End of file
```

This code, when run, outputs the following (in an 80 character wide window):

```
>> encode Input a string you would like encoded (please use only the first seven
letters, space, period, and exclamation point):Egad!  Deb bagged a cad.
encodedmessage =
  Columns 1 through 13
    1 0 0 0 1 0 1 0 0 0 1 1 0
  Columns 14 through 26
    0 0 0 0 0 0 0 0 0 0 1 1 0
  Columns 27 through 39
```

```
    1 1 1 0 1 0 1 1 0 1 1 1 0
  Columns 40 through 52
    0 0 1 0 0 0 0 0 1 0 0 0 0
  Columns 53 through 65
    0 0 0 1 0 1 1 0 1 1 0 0 0
  Columns 66 through 78
    0 0 1 0 0 0 0 0 0 0 0 0 1
  Columns 79 through 91
    1 0 0 0 0 1 1 0 0 0 0 1 0
  Columns 92 through 104
    0 0 0 0 0 1 1 0 1 1 0 1 1
  Columns 105 through 117
    0 0 0 0 0 0 0 1 1 0 1 1 0
  Columns 118 through 130
    0 0 0 1 0 0 0 0 0 0 0 0 0
  Columns 131 through 143
    0 0 1 1 0 1 1 1 0 0 0 1 1
  Columns 144 through 146
    1 1 1
```

The following is the decoder implementation. It takes a vector of 1's and 0's in the `encodedmessage` variable and outputs the alphanumeric translation of that string, according to the MIT code.

```
% Start of file, filename decode.m
% MIT Code Decoder
%

%%% Initialize variables
decodedcharacter=[];
decodedmessage=[];
codelength=0;
i=1;

messagelength=length(encodedmessage);

while i < messagelength,
  %%% Determine if it is a 6-bit or 7-bit code

  if encodedmessage(i) == 0
    codelength=6;
  else
    codelength=7;
  end

  %%% Read out the bits associated with the code
  decodedcharacter=[];
  for k=1:codelength
  decodedcharacter=cat(2,decodedcharacter,encodedmessage(i+k-1));
  end

  %%% Decode the character
  decodedcharacter=bi2de(decodedcharacter, 'left-msb');
  switch decodedcharacter
    case 0
      decodedcharacter='a';
    case 1
      decodedcharacter='b';
    case 2
      decodedcharacter='c';
    case 3
      decodedcharacter='d';
    case 4
      decodedcharacter='e';
    case 5
      decodedcharacter='f';
    case 6
      decodedcharacter='g';
    case 27
      decodedcharacter=' ';
    case 28
      decodedcharacter='.';
    case 29
      decodedcharacter='!';
    case 65
      decodedcharacter='A';
    case 66
      decodedcharacter='B';
    case 67
      decodedcharacter='C';
    case 68
      decodedcharacter='D';
    case 69
      decodedcharacter='E';
    case 70
      decodedcharacter='F';
    case 71
      decodedcharacter='G';
    case(31)
      decodedcharacter='(ETX)';
    otherwise
      decodedcharacter=' ';
  end
  decodedmessage=cat(2,decodedmessage,decodedcharacter);

  %%% Increment the position counter for the encoded message
  i=i+codelength;

end

%%% Output decoded message
fprintf(decodedmessage)
fprintf('\n')
```

```
%%% End of file
```

The ‘decode’ program, when presented with the input “Egad! Deb bagged a cad.” encoded by ‘encode’, will output the following:

```
>> decode
Egad!  Deb bagged a cad.(ETX)
```