MAE 552
Heuristic Optimization

Instructor: John Eddy
Lecture #15
2/27/02
Bit Manipulation
Bitwise operations for use in GA

*If you decide to use binary encoding!!*

Bitwise operators:

- and
- or
- xor
- left, right shift
- ones compliment
Bitwise operations for use in GA

Why “Bitwise”:

The operators are carried out individually on each set of bits in the same locus on the two args. We will see examples.
Bitwise operations for use in GA

Truth Tables:

<table>
<thead>
<tr>
<th>AND</th>
<th>X</th>
<th>Y</th>
<th>&amp;</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

| OR  | X | Y | | |
|-----|---|---|---|
| 0   | 0 | 0 | 0 |
| 0   | 1 | 1 | 1 |
| 1   | 0 | 1 | 1 |
| 1   | 1 | 1 | 1 |

<table>
<thead>
<tr>
<th>XOR</th>
<th>X</th>
<th>Y</th>
<th>^</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NOT</th>
<th>X</th>
<th>~</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Bitwise operations for use in GA

Shift Operators (<<, >>):

written as: number << n, number >> n
Move all the bits left or right by n places and fill in anywhere necessary with 0’s, truncate in the case of overrun.

example:
10 << 2 = 1010 << 2 = 101000 ( = 40 )
10 >> 2 = 1010 >> 2 = 0010 ( = 2 )
Bitwise operations for use in GA

Shift Operators (<<, >>):

For what we use them for, they are not really needed. We will only use them for moving 1’s back and forth as in:

1 << n

which is the same as $2^n$. 
Bitwise operations for use in GA

Example - using and.

\[
28 \& 9 \quad == \quad 11100 \\
& 01001 \\
\]

\[
-------------------
01000 \quad == \quad 8
\]
Bitwise operations for use in GA

Example - using or.

\[ 28 | 9 = 11100 \]
\[ | 01001 \]
\[ | 01001 \]
\[ ------------- \]
\[ 11101 \]
\[ == 29 \]
Bitwise operations for use in GA

Example - using xor.

\[ 28 \ ^\ 9 \quad == \quad 1 \ 1 \ 1 \ 0 \ 0 \]

\[ \begin{array}{c}
| 0 \ 1 \ 0 \ 0 \ 1 \\
\hline
1 \ 0 \ 1 \ 0 \ 1 \ 1
\end{array} \]

\[ 1 \ 0 \ 1 \ 0 \ 1 \quad == \quad 21 \]
Bitwise operations for use in GA

Example - using not for ones compliment (unary).

\[ \neg 28 \equiv \neg1 1 1 0 0 = 0 0 0 1 1 \equiv 3 \]

\[ \neg 31 \equiv \neg1 1 1 1 1 = 0 0 0 0 0 \equiv 0 \]
How to use these operators

1) To perform single point crossover (given 2, 8-bit strings)
   \[ P1 = 10101010 \text{ and } P2 = 01010101 \]

   Step 1 – Generate random crossover pt. bet 1 and 7.
   Let’s choose 4.

   Step 2 – Develop variables to extract the bits on either side of the crossover point. We’ll plan to use our “and” operator with our extraction variables.
How to use these operators

We start by generating a right side extraction variable as follows:

\[ rt\_op = (1 << cp) - 1 \]

In our case with \( cp = 4 \) we will get:

\[
(0 0 0 0 0 0 0 1 << 4) - 1 \\
= (0 0 0 1 0 0 0 0) - 1 \\
= (0 0 0 0 1 1 1 1)
\]
How to use these operators

We then move on to generate a left side extraction variable as follows:

\[ \text{lt}_\text{op} = \sim \text{rt}_\text{op} \]

As you can see, our left op is simply the ones compliment of our rt_op. It is then equal to

\[(1 1 1 1 0 0 0 0)\]
How to use these operators

Now what do we do with our extraction variables?

Keeping in mind our goal and recall that I mentioned using the “and” operator with our extraction variables, we can separate the bits on the left and right side of each parent string into temporary bit holder variables.

The process for creating a single child is as follows:
How to use these operators

temp1 = P1 & rt_op

0 1 0 1 0 1 0
& 0 0 0 0 1 1 1

----------------------
temp1 = 0 0 0 0 1 0 1 0
How to use these operators

temp2 = P2 & lt_op

0 1 0 1 0 1 0 1
& 1 1 1 1 0 0 0 0

----------------------
temp2= 0 1 0 1 0 0 0 0
How to use these operators

Now that we have extracted the bits, how do we assemble them into a child?

For this, we will use the “or” operator.

\[
\text{child 1} = \text{temp1} \mid \text{temp2};
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
| & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
| & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
\end{array}
\]

\[
\text{child 1} = 01011010
\]
How to use these operators

To create the other child, we apply the exact same steps except that we use the extraction variables in reverse.

That is, child 2 will be given by:

\[(P1 \land \text{lt\_op}) \lor (P2 \land \text{rt\_op})\]

(recall that child 1 was given by:

\[(P1 \land \text{rt\_op}) \lor (P2 \land \text{lt\_op})\] )
How to use these operators

Quick Note:

Many of the operations presented can be carried out in a single equation and a number of variables used in this example are not explicitly needed in the code (examples, temp1, temp2, lt_op (could use ~rt_op everywhere)).
How to use these operators

1) To perform Bit Mutation on a randomly selected string.
   
   We had a child like this: 0 1 0 1 1 0 1 0

   Step 1 – Randomly choose a bit to be mutated.
   We will mutate the bin in locus 4.

   Step 2 – Perform the mutation according to the equation:
   
   MutatedChild = Child ^ (1 << (mb-1))
How to use these operators

So considering our child and bit # 4, we have
mc = child ^ (1 << 3)

\[
\begin{array}{cccccccc}
  0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
  ^ & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
  ^ & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
  0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Note that anything xor’ed with 1 negates.