Problem 1

(a) Encode the circuit as a SAT problem

As written in Larrabee’s paper[2], one can transform a circuit element to a boolean equation(CNF form) using the idea that \( P = Q \) is expressed as \((P \rightarrow Q) \cdot (Q \rightarrow P)\).

<table>
<thead>
<tr>
<th>Logic</th>
<th>Input</th>
<th>Output</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>X,Y</td>
<td>Z</td>
<td>((Z + X) \cdot (Z + Y) \cdot (\overline{X} + \overline{Y} + Z))</td>
</tr>
<tr>
<td>NAND</td>
<td>X,Y</td>
<td>Z</td>
<td>((Z + X) \cdot (Z + Y) \cdot (\overline{X} + \overline{Y} + \overline{Z}))</td>
</tr>
<tr>
<td>OR</td>
<td>X,Y</td>
<td>Z</td>
<td>((Z + \overline{X}) \cdot (Z + \overline{Y}) \cdot (\overline{Z} + X + Y))</td>
</tr>
<tr>
<td>XOR</td>
<td>X,Y</td>
<td>Z</td>
<td>((\overline{X} + Y + Z) \cdot (X + \overline{Y} + Z) \cdot (\overline{X} + \overline{Y} + \overline{Z}) \cdot (X + Y + \overline{Z}))</td>
</tr>
<tr>
<td>NOT</td>
<td>X</td>
<td>Z</td>
<td>((X + Z) \cdot (\overline{X} + \overline{Z}))</td>
</tr>
</tbody>
</table>

Table 1: Translating Formulas into CNF

Using Table 1, the original circuit in Figure 1 is expressed as follows

\[
\text{res} = (L + \overline{J}) \cdot (L + \overline{K}) \cdot (\overline{L} + J + K) \\
J + G \cdot (J + H) \cdot (\overline{J} + \overline{G} + \overline{H}) \\
K + \overline{I} \cdot (K + \overline{F}) \cdot (\overline{K} + I + F) \\
(G + A) \cdot (\overline{G} + \overline{A}) \\
(\overline{H} + B) \cdot (\overline{H} \cdot C) \cdot (\overline{B} + \overline{C} + H) \\
(I + D) \cdot (I + E) \cdot (\overline{I} + \overline{D} + \overline{E})
\]
(b) Fault circuit

\[
\text{res} = (L' + \overline{I}) \cdot (L + K') \cdot (\overline{L} + J + K') \cdot (K' + \overline{F}) \cdot (K' + \overline{F}) \cdot (\overline{K'} + I' + F) \cdot (I') \cdot (J + G) \cdot (J + H) \cdot (\overline{J} + \overline{G} + H) \cdot (G + A) \cdot (\overline{G} + \overline{A}) \cdot (\overline{H} + B) \cdot (\overline{H} \cdot C) \cdot (\overline{B} + \overline{C} + H) \cdot (I + D) \cdot (I + E) \cdot (\overline{I} + \overline{D} + \overline{E})
\]

(c) Find a test input that exhibits the fault

Find a test vector is the same as finding a vector that satisfies a CNF formula of Figure 3. And the formula is as follows.

\[
\text{res} = (L' + \overline{I}) \cdot (L + K') \cdot (\overline{L} + J + K') \cdot (K' + \overline{F}) \cdot (K' + \overline{F}) \cdot (\overline{K'} + I' + F) \cdot (I') \cdot (J + G) \cdot (J + H) \cdot (\overline{J} + \overline{G} + H) \cdot (G + A) \cdot (\overline{G} + \overline{A}) \cdot (\overline{H} + B) \cdot (\overline{H} \cdot C) \cdot (\overline{B} + \overline{C} + H) \cdot (I + D) \cdot (I + E) \cdot (\overline{I} + \overline{D} + \overline{E}) \cdot (\overline{L} + L' + Z) \cdot (L + \overline{L'} + Z) \cdot (L + L' + Z) \cdot (\overline{L} + \overline{L'} + \overline{Z})
\]

- Activate - For the stuck 1 value, make an input that generates the reverse of that value. So D and E should be 1.
• Propagate and Justify - For the error to propagate, give 1 input to the AND gate and 0 to the OR gate.

Using this method the test vector has three patterns. There are three patterns because there are 3 ways to make $J$ equals 1.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Test patterns

There are two main ways to solve the SAT problems([1]).

• Resolvent Algorithm.
• Search Based Algorithm.

Resolvent algorithm uses unate variable rule and binate variable rule. By applying those rules the SAT equation can be simplified to find values that evaluate the equation to 1.

Search Based Algorithm enumerates all input assignments to evaluate them to decide whether a formula is satisfiable.

**Problem 2**

(a) **Draw retiming graph and compute W and D matrices**

Figure 4 shows the graph of the circuit.

Using this graph D(delay) matrix is in Table 3.

And W(weight) matrix is in Table 4.
Figure 4: Retiming graph

Table 3: D matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>8</td>
<td>17</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>5</td>
<td>14</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
<td>14</td>
<td>4</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td>10</td>
<td>14</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
<td>17</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4: W matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
(b) **Minimum clock cycle**

Before retiming path \( B \to D \to E \) is the critical path with delay 10.

(c) **Is retiming is possible with a clock period of 5?**

In order to retiming one should build constraints out of delay matrix. Out of D matrix components, the value that exceeds 5 should be selected to make a constraint.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>8</td>
<td>17</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td>B</td>
<td>10</td>
<td>5</td>
<td>14</td>
<td>11</td>
<td>14</td>
</tr>
<tr>
<td>C</td>
<td>9</td>
<td>14</td>
<td>4</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
<td>10</td>
<td>14</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
<td>17</td>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5: Components in D matrix that is more than 5 delays

And the constraint is expressed as \( r(u) - r(v) \leq W(u, v) - 1 \).

So, the constraint inequalities are as follows.

\[
\begin{align*}
    r(A) - r(B) & \leq 0 \\
    r(A) - r(C) & \leq 0 \\
    r(A) - r(D) & \leq 0 \\
    r(A) - r(E) & \leq 0 \\
    r(B) - r(A) & \leq 1 \\
    r(B) - r(C) & \leq -1 \\
    r(B) - r(D) & \leq 0 \\
    r(B) - r(E) & \leq 0 \\
    r(C) - r(A) & \leq 2 \\
    r(C) - r(B) & \leq 3 \\
    r(C) - r(D) & \leq -1 \\
    r(C) - r(E) & \leq 0 \\
    r(D) - r(B) & \leq 2 \\
    r(D) - r(C) & \leq 2 \\
    r(E) - r(A) & \leq 3 \\
    r(E) - r(B) & \leq 4 \\
    r(E) - r(C) & \leq 0 \\
    r(E) - r(D) & \leq 1 
\end{align*}
\]

Out of these constraint sets, we can make another graph. For example, \( r(A) - r(B) \leq 0 \) can be expressed as Figure 5.

![Figure 5: Constraint expressed in graph](image)
By using this relation, the whole graph is Fig 6.

![Constraint graph](image_url)

**Figure 6: Constraint graph**

Finally, using Bellman-Ford algorithm, one can solve this problem to find out how the registers are moved. I implemented a Bellman-Ford algorithm to solve this problem using Python programming language. The source code is listed in page 20.

Using the BF solver the result is in Table 11.

<table>
<thead>
<tr>
<th>r(A)</th>
<th>r(B)</th>
<th>r(C)</th>
<th>r(D)</th>
<th>r(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6: Result from BF solver

**Problem 3**

(a) Make BDD for 2 bit comparator

Table 7 shows the truth table for 2 bit comparator of \( b > a \). Figure 7 shows BDD if ordered \( a_1 < b_1 < a_0 < b_0 \) and its simplification. The number of non-terminal is 5. Figure 8 shows BDD if ordered \( a_1 < a_0 < b_1 < b_0 \) and its simplification. The number of non-terminal is 9. In both cases, the dotted line implies 0 and solid line implies 1.

(b) BDD for n bit comparator

As one can see by rearranging the Table 7 into Table 8, If the sequence is ordered \( a_{n-1} < b_{n-1} < \cdots < a_0 < b_0 \). The 0’s and 1’s are collected in one parts so that the resulting BDD can be simplified. It is because the MSB is compared first, so the LSB’s are always 0 if MSB of \( b \) is smaller than that of \( a \) and vice versa. So the circuit grows in linear fashion.

But if the sequence is ordered as \( a_{n-1} < a_{n-2} < \cdots < b_1 < b_0 \), the bits are shuffled so that the circuit cannot be simplified. So the circuit grows in exponential fashion.

(c) Implement XOR with BDD

The equation is \( x \oplus y = ITE(x, y', y) \) because of the relation \( x \oplus y = x \cdot y' + x' \cdot y \) and \( ITE(f, x, y) = x \cdot f + x' \cdot y \). So only 1 \( ITE \) operation is needed to construct XOR function.
Table 7: Truth Table for 2bit Comparator

<table>
<thead>
<tr>
<th>a1</th>
<th>a0</th>
<th>b1</th>
<th>b0</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 7: 2 bit comparator BDD and its simplification - case 1
Figure 8: 2 bit comparator BDD and its simplification - case2

<table>
<thead>
<tr>
<th>a1</th>
<th>b1</th>
<th>a0</th>
<th>b0</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 8: Truth Table for 2bit Comparator when a and b are alternated
(d) Prove ITE decomposition

Let $Z = ITE(f, g, h)$ and let $x$ be the top variable of function $f$, $g$ and $h$.

\[
Z = x \cdot Z_x + x' \cdot Z_x',
\]
\[
= x(f \cdot g + f' \cdot h)_x + x'(f \cdot g + f' \cdot h)_x',
\]
\[
= x(f_x \cdot g_x + f'_x \cdot h_x) + x'(f_x \cdot g_x + f'_x \cdot h_x')
\]
\[
= ITE(x, ITE(f_x, g_x, h_x), ITE(f'_x, g'_x, h'_x))
\]

So, $ITE(f, g, h) = x \cdot ITE(f_x, g_x, h_x) + x' \cdot ITE(f'_x, g'_x, h'_x)$

**Problem 4**

(a) Write down $\delta$ for the state machine

The transition $\delta$ can be expressed as follows.

\[
\delta(a, b, c, d, x, a^+, b^+, c^+, d^+) = a \overline{b} \overline{c} \overline{d} \overline{x} \overline{a^+} \overline{b^+} \overline{c^+} \overline{d^+} + \\
\overline{a} b c \overline{d} x a^+ b^+ c^+ \overline{d^+} + \\
\overline{a} b \overline{c} \overline{d} x a^+ b^+ c^+ d^+ + \\
\overline{a} b \overline{c} d 1 a^+ b^+ c^+ \overline{d^+} + \\
a b \overline{c} \overline{d} x a^+ b^+ c^+ \overline{d^+} + \\
\overline{a} b \overline{c} \overline{d} 1 a^+ b^+ c^+ d^+
\]

I tried to simplify the logic using Espresso, but this logic cannot be simplified further. And the transition can be expressed in functional representation as follows.

\[
R_{k+1} = \delta(R_k, x)
\]

The other way to express the transition is relational representation as follows.

\[
T(R_{k+1}, R_k, x) = \begin{cases} 
1, & R_{k+1} = \delta(R_k, x) \\
0, & \text{otherwise}
\end{cases}
\]

(b) Deriving $R_1$, $R_2$ and so on

Using the expressions one can get the next state.

\[
a^+ = d
\]
\[
b^+ = a \overline{x} + b \overline{x}
\]
\[
c^+ = a x + c
\]
\[
d^+ = b x
\]

So the $T$ can be expressed as follows.

\[
T = (a, b, c, d, x, a^+, b^+, c^+, d^+)(a^+ \oplus d)(\overline{b^+} \oplus (a \overline{x} + b \overline{x}))(c^+ \oplus (a x + c))(d^+ \oplus b x)
\]
This equation checks, when there is a input \( x \), if the result state is the same is expected. When the state is \( R_0 \) and the input is 0 the \( T \) is as follows.

\[
R_1 = T(1, 0, 0, 0, a^+, b^+, c^+, d^+) = (a^+ \oplus 0)(b^+ \oplus 1)(c^+ \oplus 0)(d^+ \oplus 0)
\]

This equation means that \( a^+b^+c^+d^+ \) equals 0100. Again one can use the same method to get the \( R_2 \). In this case input \( x \) is 1 to go to the other state.

\[
R_2 = T(0, 1, 0, 0, a^+, b^+, c^+, d^+) = (a^+ \oplus 0)(b^+ \oplus 0)(c^+ \oplus 0)(d^+ \oplus 1)
\]

(c) Working for VeriBackward

What the boss wants to express can be drawn by Figure 9 and Figure 10. As can be seen in the 9, one can progress from normal state to error state. It can be seen as described in Fig 10. That is, from the error state, one can go to the next step on and on. And if the state includes the initial state, the circuits are not equivalent.

With this method when we iterate the state, the state goes to the initial state which means stop. And I can check whether \( R_0 \) is contained in \( E_m \) by using the following 1 sentence.

With Initial state \( I(s) \), and a fault state \( F(s) \), if \( (S \cdot F) \neq 0 \), \( R_0 \) is contained in \( E_m \).

![Figure 9: State Transition to Error State](image)

**Problem 5**

(a) Find technology mapping

If one maps the logic gate to the circuit one by one, one can get the area of 15.0 as is seen in Table 9.

My algorithm for finding the best match is as follows.
Figure 10: State Transition from Error State

Figure 11: Result of the 1:1 mapping

Table 9: 1:1 mapping result
Step 1. Transform each library in NAND2 form.
Step 2. Transform each logic gate in NAND2 form and normalize.
Step 3. If there is a fan-out isolate the circuit
Step 4. Calculate the circuit
for each logic
loop
  Mapping each circuit to minimize the area
  Calculate at each step
end loop
end each circuit
Step 5. attach the isolated circuit to each circuit and go to for loop and recalculate
Step 6. If there is no not gate between the gate insert the 2 not gate and recalculate

**Step1**
For Step1 the library circuit should be transformed into orthogonal AND circuit. The transform rule is shown in Figure 12.

![Transform the library](image12)

**Step2**
For Step2 the input circuit is transformed into orthogonal AND circuit. The transformed result is shown in Figure 13.

**Step3**
For Step3 If the circuit has fan-out separate the circuit. The transformed result is shown in Figure 14.

**Step6**
For Step6, if there is no circuit between the gates, insert 2 not gates to see if it is possible to transform into AOI or OAI circuit. The transformed result is shown in Figure 15.
Figure 13: Transform the circuit

Figure 14: Remove the fan-out

Figure 15: Insert 2 not gates
(b) Result

With the example circuit, after going through all the algorithm and compare the result, one can get the following result. Mapped result is in Figure 16 and calculated size is in Table 10. In this case, by inserting 2 not gates and separating the circuit into 3 parts, I can get the best result.

Problem 6

(a) Survey

Before solving the problem, it is necessary to see when those four cases occur. I used the book [4] and [3] as a reference.

1. Blocked

![](blocked_path.png)

Figure 17: Blocked path

The path is blocked when one of the gate in the path sends only 1 or 0. For example, if AND gate has input 0, it always sends 0. And if OR gate has input 1, it sends 1 always. In both cases the path is blocked. Figure 17 shows the path that is blocked. If $b = 0$ path $b \rightarrow e \rightarrow z$ is blocked, where as path $a \rightarrow c \rightarrow d \rightarrow y$ becomes critical path.

2. Statically co-sensitizable

Consider a path $P = (v_{s0}, v_{s1}, \ldots, v_{sm})$. A vector is statically co-sensitizes a path to 1 (or to 0) if input $x_m = 1$ (or 0) and if $v_{s_{m-1}}$ has a controlling value whenever $v_{s_{m}}$ has a controlled value.

As an example with Figure 18, Path $a \rightarrow d \rightarrow g \rightarrow o$ is statically co-sensitizable to 0 by input $a = 0, b = 0, c = 0$. It is because the gates with controlled value(o, d) have a controlling input along the path ($g = 0, a = 0$).

3. Statically sensitizable

Figure 19 shows a statically sensitizable path with test vector $b = 1, c = 1, d = 1$. The algorithm for finding test vector is as follows. Actually, it is same as ‘Fault propagation’ algorithm for finding
Figure 18: Statically co-sensitizable path

stuck-at error.

Step 1. Identify the path.
Step 2. Identify input vector.
   Step 2-1. With AND gate, give 1 to make a path. Find vector to send 1 from the previous gate.
   Step 2-2. With OR gate, give 0 to make a path. Find vector to send 0 from the previous gate.

At Figure 19, o is AND gate. So the input is 1. In order to send 1 from g, input vector should be 1.

Figure 19: Statically sensitizable path

HFRPDFT

Statically sensitizable circuit can be simplified with the following rule.

- With AND gate, if the input is 1, the gate becomes buffer.
- With OR gate, if the input is 0, the gate becomes buffer.

After simplification, if the path to the input gate has the same delay. We can identify the path as HFRPDFT. Figure 20 shows 2 circuits after simplification. Circuit in upper part is not HFRPDFT but circuit in lower part is HFRPDFT. The HFRPDFT path has 2 test vectors, the first vector makes result 0 and the second vector makes the result 1.

(b) Identify each path

Path 1, 2, 3, 4, 5

Figure 21 shows path 1, 2, 3, 4, 5 and its simplified circuit when input vector is $d = 0$ to propagate the signal through the path. The five paths are blocked.

Path 6, 7

Figure 22 shows path 6, 7 when input vector is $d = 0$ to propagate the signal through the path. The two paths are blocked.
Figure 20: After simplification

Figure 21: Blocked path

Figure 22: Path 6,7
Path 8
Figure 23 shows path 8 and its simplified circuit when input vector is \(a = 0, b = 1, c = 0\) to propagate the signal through the path.

Path 9
Figure 24 shows path 9 and its simplified circuit when input vector is \(b = 1, c = 1\) to propagate the signal through the path.

Path 10
Figure 25 shows path 10 and its simplified circuit when input vector is \(a = 0, b = 0, d = 1\) to propagate the signal through the path. This path is HFRPFDFT with test vector \(<(0, 0, 0, 1), (0, 0, 1, 1)\>\).

Path 11
Figure 26 shows path 11 and its simplified circuit when input vector is \(a = 0, b = 1, d = 1\) to propagate the signal through the path.

Path 12
Figure 27 shows path 12 and its simplified circuit when input vector is \(a = 1, c = 0, d = 1\) to propagate the signal through the path. This path is HFRPFDFT with test vector \(<(1, 0, 0, 1), (1, 1, 0, 1)\>\).

Path 13
Figure 28 shows path 13 and its simplified circuit when input vector is \(a = 0, b = 1, d = 0\) to propagate the signal through the path.

(c) Result
Table 11 shows the identified path.
Figure 24: Path 9

Figure 25: Path 10

<table>
<thead>
<tr>
<th>Path</th>
<th>Condition</th>
<th>Test Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,3,4,5,6,7</td>
<td>Blocked</td>
<td></td>
</tr>
<tr>
<td>8, 9,11,13</td>
<td>Statically sesitizable</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>HFRPDFT</td>
<td>&lt; (0, 0, 0, 1), (0, 0, 1, 1) &gt;</td>
</tr>
<tr>
<td>12</td>
<td>HFRPDFT</td>
<td>&lt; (1, 0, 0, 1), (1, 1, 0, 1) &gt;</td>
</tr>
</tbody>
</table>

Table 11: Result
Figure 26: Path 11

Figure 27: Path 12
A. Bellman-Ford algorithm implementation

```python
# maxpath BIG <
# minpath SMALL >
-INF__ = 400000
class Vertex:
    def __init__(self, number):
        self.number = number;
        self.list = []
    def add(self, edge):
        self.list.append(edge);
    def printme(self):
        print "-------------------\n";
        for edge in self.list:
            edge.printme();
        print "-------------------\n";

class VertexList:
    def __init__(self):
        self.list = {}
    def add(self, name, vertex):
        self.list[name] = vertex;
    def length(self):
        return len(self.list);
    def __getitem__(self, name):
        return self.list[name];

class Edge:
    def __init__(self, start, stop, weight):
        self.start = start;
        self.stop = stop;
```

Figure 28: Path 13
30       self.weight = weight;
31     def printme(self):
32         print "START:%d STOP:%d - WEIGHT:%d\n" % (self.start, self.stop, self.
33                                         weight);
34     
35     def BF(vertexList):
36         number = vertexList.length();
37         distance = {}
38         
39         # initialize
40         for x in range(number):
41             distance[x] = _INF_-;
42             distance[0] = 0;
43             count = 0;
44             
45         # First
46         v = vertexList['v0'];
47         S1 = [];
48         S1.append(v);
49         S2 = [];
50         
51         while (count <= number and len(S1) > 0):
52             # Get every vertex
53             for vertex in S1:
54                 # get every edge connected with the vertex
55                 for edge in vertex.list:
56                     vj = "v%d" % edge.stop;
57                     if (distance[edge.stop] > distance[edge.start] + edge.weight):
58                         distance[edge.stop] = distance[edge.start] + edge.weight;
59                         S2.append(vertexList[vj])
60                 
61             S1 = S2;
62             S2 = [];
63             count += 1
64             printlist(distance);
65             print("\n");
66             
67         if (count > number):
68             print("ERROR: Positive cycle");
69             
70             return distance
71     
72     def printlist(list_):
73         val = "";
74         for i in list_:
75             val = "%s %d" % (val, list_[i])
76         print val
77         
78     
79     if __name__ == "__main__":
80         # Edge
81         e1 = Edge(1,0,0);
82         e2 = Edge(2,0,0);
83         e3 = Edge(3,0,0);
84         e4 = Edge(4,0,0);
84  e5 = Edge(0,1,1);
85  e6 = Edge(2,1,-1);
86  e7 = Edge(3,1,0);
87  e8 = Edge(4,1,0);
88
89  e9 = Edge(0,2,2);
90  e10 = Edge(1,2,3);
91  e11 = Edge(3,2,-1);
92  e12 = Edge(4,2,0);
93
94  e13 = Edge(1,3,2);
95  e14 = Edge(2,3,2);
96
97  e15 = Edge(0,4,3);
98  e16 = Edge(1,4,4);
99  e17 = Edge(2,4,0);
100 e18 = Edge(3,4,1);
101
102  # Vertex
103  v0 = Vertex(0); #A
104  v0.add(e5); v0.add(e9); v0.add(e15);
105  v1 = Vertex(1); #B
106  v1.add(e1); v1.add(e10); v1.add(e13); v1.add(e16);
107  v2 = Vertex(2); #C
108  v2.add(e2); v2.add(e6); v2.add(e14); v2.add(e17);
109  v3 = Vertex(3); #D
110  v3.add(e3); v3.add(e7); v3.add(e11); v3.add(e18);
111  v4 = Vertex(4); #E
112  v4.add(e4); v4.add(e8); v4.add(e12);
113
114  vlist = VertexList();
115  vlist.add('v0', v0);
116  vlist.add('v1', v1);
117  vlist.add('v2', v2);
118  vlist.add('v3', v3);
119  vlist.add('v4', v4);
120
121  res = BF(vlist);
122  print(res);
References


