On Path Selection In Combinational Logic Circuits

Wing Ning Li+, Sudhakar M. Reddy++, Sartaj Sahni+

ABSTRACT

In order to ascertain correct operation of digital logic circuits it is necessary to verify correct functional operation as well as correct operation at desired clock rates. To ascertain correct operation at desired clock rates signal propagation delays along a set of selected paths are verified to fall within allowed limits by applying appropriate stimuli. Earlier it was suggested that an appropriate set of paths to test would be the one that includes at least one path, with maximum modeled delay, for each circuit lead or gate input. In this paper, algorithms to select such sets of paths with minimum cardinality are given.

KEYWORDS and PHRASES

Testing, combinational circuits.

+Department of Computer Science, University of Minnesota, Minneapolis, MN 55455. Research supported by the National Science Foundation under grants DRC84-20935 and MIP86-17374.
++Department of Electrical and Computer Engineering, University of Iowa, Iowa City, IA 52242. Research supported by Texas Instruments, Inc. and by SDIO/IST contract No. N00014-87k-0419 managed by U.S. Office of Naval Research.

1 INTRODUCTION

In order to ascertain correct operation of logic circuits it is necessary to verify correct functional operation as well as correct operation at intended clock rates [1-4]. In order to verify correct operation at desired clock speeds often two approaches are proposed. One is to test, through what are called delay tests, correct signal propagation along every path in the logic circuit under test or at least every path whose modeled delay is $\alpha T_{\max}$, where $T_{\max}$ is the maximum modeled delay in the circuit under test [8]. This approach is often impractical due to the large number of paths in logic circuits encountered in practice. The second approach is to select a set of paths, say SP, in the logic circuit such that for each lead l in the given circuit C there is at least one path in SP which exhibits maximum modeled delay among all circuit paths that contain l [2]. We give procedures to select a minimum number of paths to meet this latter objective.

In Section 2, we develop the terminology to be used in the paper. In addition a formal graph model for the path seletion problem is developed. A polynomial time algorithm to find a minimum cardinality path set is developed in Sections 3 through 6. This result is somewhat surprising as our problem is a path covering problem and most other covering problems are known to be $NP-$ complete. Two heuristics that are faster than the minimum cardinality algorithm are presented in Sections 7 and 8. In Section 9, we present experimental results comparing the performance of the minimum cardinality algorithm and the two heuristics.

2 TERMINOLOGY

In this paper we consider the problem of selecting a minimum number of paths from inputs to outputs of *combinational logic circuits* such that each circuit lead (gate input) l is, (i) included in at least one selected path p and (ii) the modeled signal propagation delay along path p is maximum among all paths that contain l. We assume that the combinational logic circuit is constructed of AND, OR, NAND, NOR and NOT gates. Associated with each gate input is a propagation delay for rising (i.e. 0 to 1) and possibly different delay for falling (i.e. 1 to 0) transition [4].

In order to develop procedures to select a minimum number of paths as defined above, we model the combinational logic circuit as a directed graph. A gate in the logic circuit is represented by a node and a gate input by a directed edge into the node representing the gate. In the graph model there are also nodes representing primary inputs (called source nodes) and circuit outputs (called sink nodes). Details of the graph model are developed below.

A *circuit graph*, (abbreviated to *circuit*) C, is a four tuple (V,E,f_V,f_E) where:

a) V = vertex set

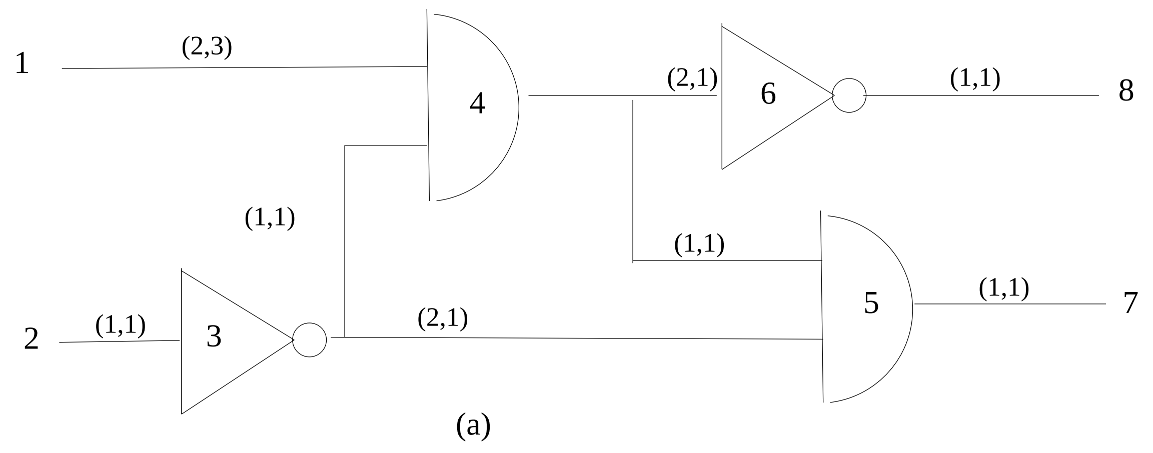
b) E = edge set

c) (V,E) is a directed acyclic graph (dag)

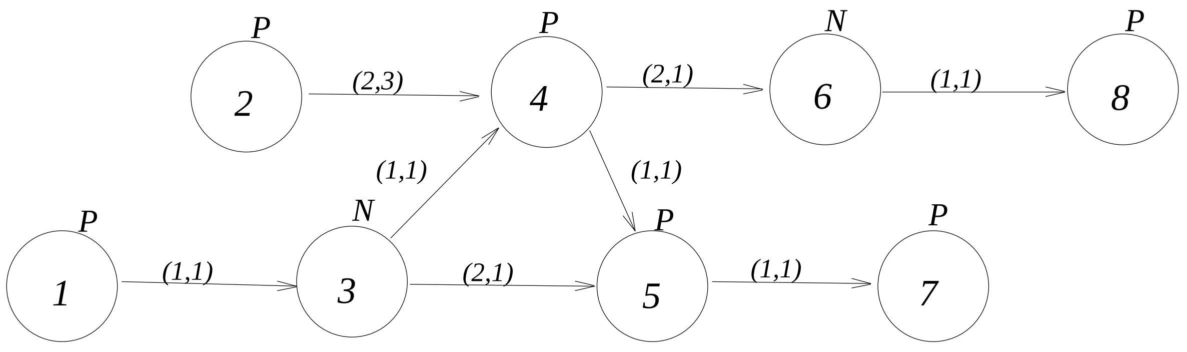
d) f_V is a labeling function with domain V and range $\{P,N\}$. $f_V(v) = N$ if the circuit element corresponding to vertex v inverts the input signal transition from rising to falling or from falling to rising. $f_V(v) = P$ if the input signal transition is not inverted. For example, 'and' and 'or' gates do not invert the input signal transition while 'not' and 'nand' gates do.

e) f_E is an edge labeling function with domain E and range $R^+ \times R^+$ where R^+ is the set of positive real numbers. $f_E(<i,j>) = (d_r, d_f)$ where d_r is the rising delay for edge $<i,j>$ and d_f is its falling delay.

An example combinational circuit and its corresponding circuit graph are shown in Figure 1. Let $C = (V, E, f_V, f_E)$ be a circuit. A vertex $v \varepsilon V$ is a source vertex iff its in-degree is 0. It is a sink vertex iff its out-degree is 0. For the example graph of Figure 1, the source vertices are 1 and 2 and the sinks are 7 and 8. Let $L = v_1, v_2, ..., v_k$ be a directed path in C that begins at a source vertex and ends at a sink vertex. Let S denote the signal type at the source vertex v_1. S is either a rising signal (R) or a falling signal (F). Let (L,S) be a path, source signal pair. The delay associated with (L,S) is the sum of the delays of the edges on L. In obtaining this sum, the rising delay of an edge is used if the signal through it is rising. The falling delay is used otherwise. As an example, consider the path $L = 1,3,4,6,8$ in the circuit of Figure 1. The delay for (L,R) is 5 and that for (L,F) is 4.



(a)



(b)

Figure 1: An example circuit.

The pair (L,S) rising (falling) *covers* the edge $<i,j>$ iff

i) $<i,j>$ is an edge of L

ii) the signal through $<i,j>$ is rising (falling)

iii) the delay associated with (L,S) is maximum amongst all path, source signal pairs (L,S) that satisfy i) and ii).

In the circuit of Figure 1, the pair (L,S) where $L = 1,3,5,7$ and $S = R$ falling covers the edges <3,5>, whereas the pair (L,S) with $L = 1,3,5,7$ and $S = F$ falling covers the edge <1,3> and rising covers the edges <3,5> and <5,7>.

The *circuit cover* problem is to find the minimum number of pairs (L,S) such that each edge of the circuit is rising and falling covered by at least one of these pairs. A minimum cover for the circuit of Figure 1 is (L_1,F), (L_2,R), (L_3,R), (L_3,F), (L_4,F), (L_5,R) and (L_5,F) where $L_1 = 1,3,4,5,7$, $L_2 = 1,3,4,6,8$, $L_3 = 1,3,5,7$, $L_4 = 2,4,5,7$ and $L_5 = 2,4,6,8$.

3 FIRST SIMPLIFYING TRANSFORMATION

A *network*, N, is a three tuple (U,A,w_A) where:

a) U = vertex set

b) A = edge set

c) (U,A) is a directed acyclic graph

d) w_A is an edge weighting function with domain A and range R^+.

The terms source and sink are defined as in the case of circuits. The weight of a source to sink path L is the sum of the weights of the edges in L. The path L *covers* the edge $<i,j>$ iff:

i) $<i,j>$ is an edge of L

ii) The weight of L is maximum amongst all paths L that contain edge $<i,j>$.

The *network cover* problem is to find the minimum number of paths such that each edge of the network is covered by at least one of these paths.

Figure 2 shows an example network. The source vertices are 1 and 2 and the sinks are 7 and 8. The path $L = 1,3,4,6,8$ covers the edges <1,3> and <3,4>. The paths $L_1 = 1,3,4,6,8$, $L_2 = 1,3,5,7$, $L_3 = 2,4,5,7$ and $L_4 = 2,4,6,8$ form a minimum network cover.

Every circuit cover problem may be transformed into an equivalent network cover problem as below:

Transformation 1

Step1: Replace each vertex $v \varepsilon V$ of the circuit (V,E,f_V,f_E) by a pair of vertices v_R and v_F. v_R (v_F) corresponds to the case where the incoming signal is rising (falling).

Step2: Replace each edge $<u,v>$ in E with a pair of edges as below:

(i) if $f_V(u) = P$, then replace $<u,v>$ with $<u_R,v_R>$ and $<u_F,v_F>$.

(ii) if $f_V(u) = N$, then replace $<u,v>$ with $<u_R,v_F>$ and $<u_F,v_R>$.

Step3: The weight of the edges $<u_R,v_R>$ and $<u_F,v_R>$ is the rising delay of $<u,v>$ and for the

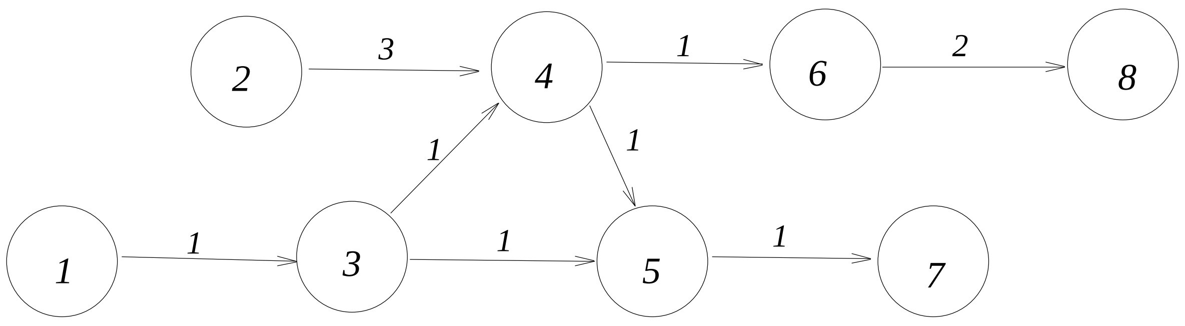


Figure 2: An example network.

edges

$<u_F, v_F>$ and $<u_R, v_F>$ is the falling delay of $<u, v>$.

Figure 3 shows the network that results when the above transformation is applied to the circuit of figure 1.

Theorem 1: Let C be a circuit and N the network obtained using the above transformation.

(a) Every pair (L,S) where L is a source to sink path of C and S is a signal type corresponds to a unique source to sink path in N and vice versa.

(b) The delay associated with (L,S) equals the weight of the corresponding path in N.

(c) Every minimum cover of C corresponds to a minimum cover of N and vice versa. The corresponding cover in N is obtained by replacing each (L,S) in the cover for C by its corresponding source to sink path in N.

Proof: Follows directly from the transformation process from C to N.□

As a result of Theorem 1, we may obtain a minimum cover for C as follows:

1. Construct N using the above transformation
2. Obtain a minimum cover for N
3. Obtain the corresponding (L,S) pairs for this cover.

4 SECOND SIMPLIFYING TRANSFORMATION

Let $G = (W,X)$ be a directed acyclic graph. W is the vertex set and X the edge set. A path L covers the edge $<i,j>$ iff $<i,j>$ is on the path. The *dag cover* problem is to find the minimum number of source to sink paths such that each edge is covered by at least one path. Figure 4 shows an example dag. A minimum path set that covers all the edges is: $L_1 = 1,5,6,8,10$, $L_2 = 2,5,7,9,11$,

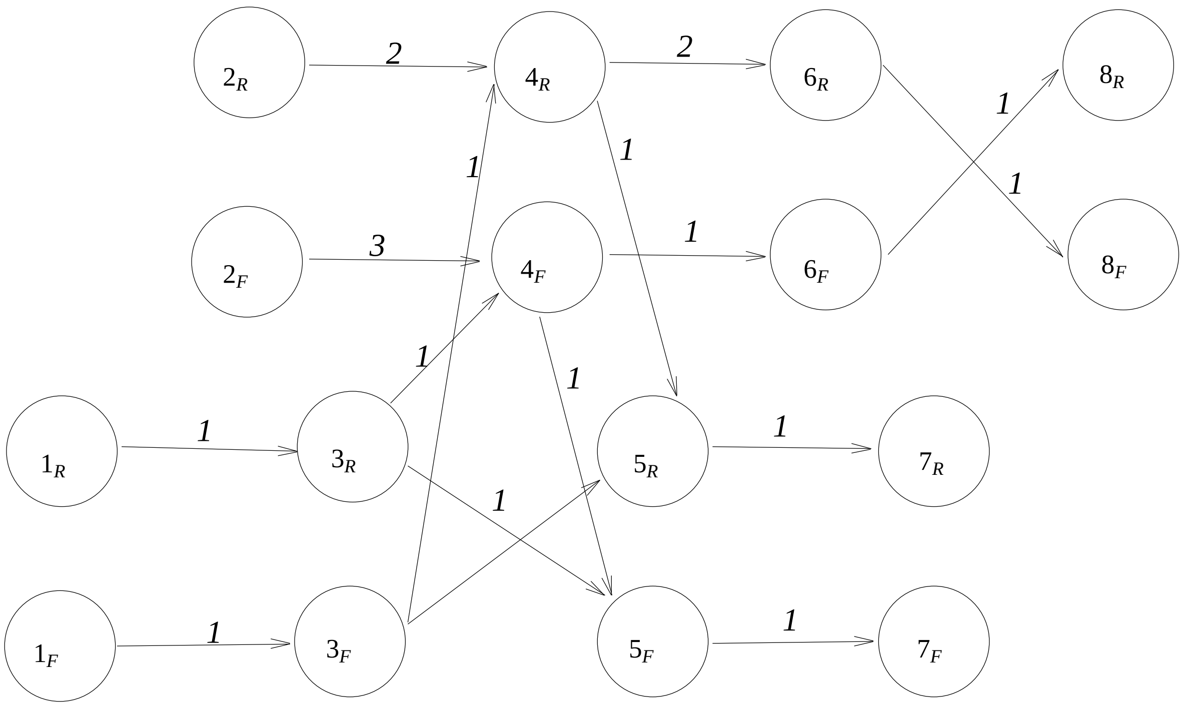


Figure 3: Network corresponding to circuit of Figure 1.

L_3 = 3,5,7,9,12 and L_4 = 4,5,7,9,13.

In this section, we show how any network cover instance N may be transformed into a dag cover instance G such that the number of paths in the dag cover equals the number of paths in the network cover. Furthermore, the network cover is easily obtained from the dag cover.

The *length* of a path in a network N is the sum of the weights of the edges on the path. We shall use the terms length and weight interchangeably in this paper. Let $source(j)$ be the set of all paths that begin at a source vertex and end at vertex j; Let $sink(i)$ be the set of all paths that begin at vertex i and end at a sink vertex. Let $longest(X)$ be the set of longest paths in set X. Note that all paths in $longest(X)$ have the same length. Every edge, $<i,j>$, in the network N may be classified as below:

1) type yy —— $<i,j>$ is on a path in $longest(sink(i))$ and $longest(source(j))$

2) type ny —— $<i,j>$ is not on any path in $longest(sink(i))$ but is on a path in $longest(source(j))$

3) type yn —— $<i,j>$ is on a path in $longest(sink(i))$ but not on any path in $longest(source(j))$

4) type nn —— $<i,j>$ is not on any path in $longest(sink(i))$ or $longest(source(j))$.

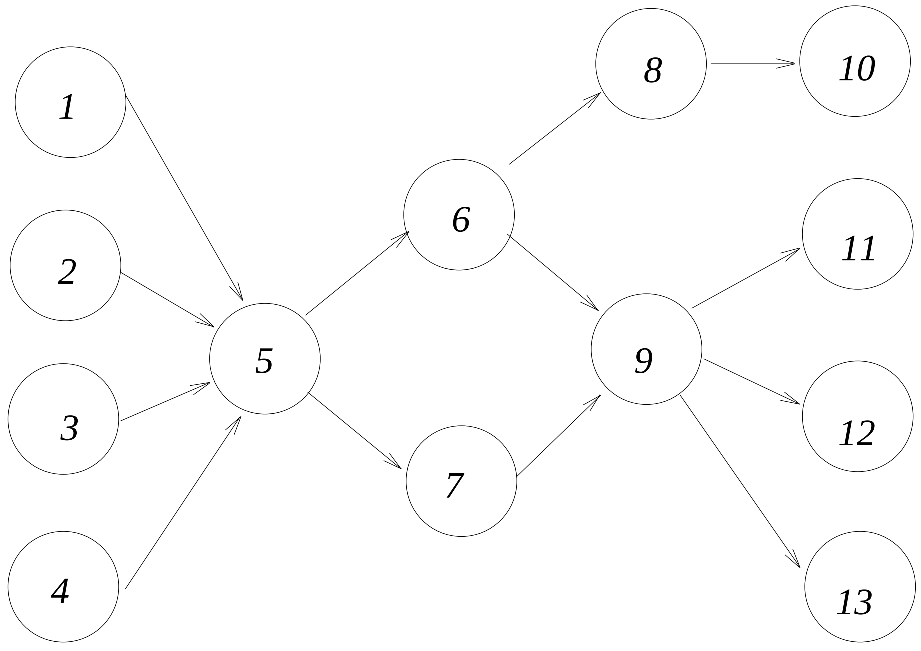


Figure 4: An example dag.

The edges of the network of Figure 3 are classified in Figure 5.

Observation 1: Let v be any vertex of N that is not a sink. There is at least one edge $<v,j>$ of type yy or yn. □

Observation 2: Let v be any vertex of N that is not a source. There is at least one edge $<i,v>$ of type yy or ny. □

Lemma 1: Let $<i,j>$ be a type nn edge of N. $<i,j>$ cannot be on a path L that covers a different edge $<u,v>$.

Proof: $<i,j>$ cannot be on a longest path that includes $<u,v>$. To see this, note that if $<u,v>$ precedes $<i,j>$ in L, then we may replace the tail of L beginning at i by a path in $longest(sink(i))$ to get a path longer than L that includes $<u,v>$. If $<i,j>$ precedes $<u,v>$, then we may replace the prefix of L up to and including edge $<i,j>$ by a path in $longest(source(j))$ to get a path longer than L that includes $<u,v>$. Hence, L cannot cover $<u,v>$. □

Lemma 2: Let $<i,j>$ be a type yn edge of N. A path L that contains $<i,j>$ cannot cover any of the edges that follow $<i,j>$ on this path.

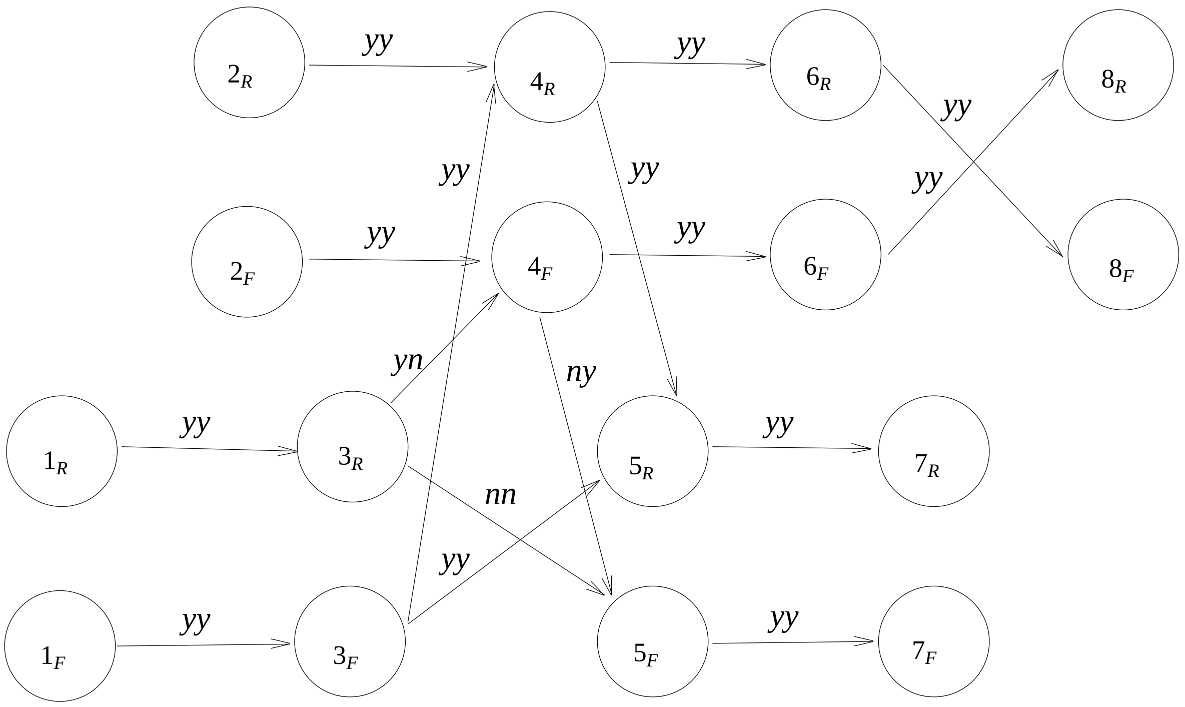


Figure 5: Network of Figure 3 with edges classified.

Proof: Similar to that of Lemma 1. □

Lemma 3: Let $<i,j>$ be a type *ny* edge of N. A path L that contains $<i,j>$ cannot cover any of the edges that precede $<i,j>$ on this path.

Proof: Similar to that of Lemma 1. □

Lemma 4: Let $cover(N)$ be a network cover of N. There is no path L in $cover(N)$ such that L has two or more edges of type *nn*.

Proof: From Lemma 1, it follows that if there is such an L in $cover(N)$, then L can cover no edge of N. Hence $cover(N) - \{L\}$ is a smaller set of paths that covers all edges of N. So, $cover(N)$ is not a network cover. This contradicts the definition of $cover(N)$. □

Lemma 5: Let $cover(N)$ be a network cover of N. There is no path $L \varepsilon\, cover(N)$ that has a type *yn* edge that precedes a type *ny* edge.

Proof: From Lemmas 2 and 3, it follows that such a path would cover no edges. Hence $cover(N) - \{L\}$ would be a smaller path set that covers all edges. □

Let $M \subseteq cover(N)$ be the (possible empty) subset of paths that contain an edge of type *nn*. From Lemma 5, it follows that every path $L \varepsilon\, cover(N) - M$ may be written as $L_L L_M L_R$ where L_L is the left part, L_M the middle part, and L_R the right part. L_M contains only *yy* edges, L_L contains *yy* and *ny* edges (the last edge in L_L is of type *ny*, L_L is empty if L has no *ny* edge), and L_R contains *yy* and *yn* edges (the first edge in L_R is of type *yn*, L_R is empty if L has no *yn* edge).

Lemma 6: Let $L = L_L L_M L_R$ as above. L covers exactly those edges in L_M, the last edge in L_L (if L_L is not empty), and the first edge in L_R (if L_R is not empty).

Proof: We need to show that L is a longest path containing any of these edges. Furthermore, L is not a longest path containing the remaining edges. The first of these follows from the edge classification scheme. The second follows from Lemmas 2 and 3. □

Lemma 7: Every path L that can be written as $L_L L_M L_R$ as above covers exactly those edges identified in Lemma 6.

Proof: Same as for Lemma 6. □

The preceding Lemmas motivate the following transformation of a network N into a dag G.

Transformation 2

Each edge $<i,j>$ of type *nn*, *yn*, *ny* is replaced by a new edge as given in the table of Figure 6.

edge type	new edge	new vertex
nn	$<l_{ij}, r_{ij}>$	l_{ij}, r_{ij}
yn	$<i, r_{ij}>$	r_{ij}
ny	$<l_{ij}, j>$	l_{ij}

Figure 6: Replacement for edge $<i,j>$

Figure 7 shows the dag that results when Transformation 2 is applied to the network of Figure 3. Figure 8(a) shows a dag cover. The paths in this dag cover are easily mapped to paths in the original network. These are shown in Figure 8(b). These network paths are close to being a network cover. The paths need to be extended so they begin at a network source and end at a sink.

Theorem 2: Let N be a network and $G = T2(N)$ be the dag obtained using Transformation 2 on N. Let $cover(N)$ and $cover(G)$, respectively, be a network cover of N and a dag cover of G. Let $|cover()|$ denote the number of paths in the *cover*. $|cover(N)| = |cover(G)|$.

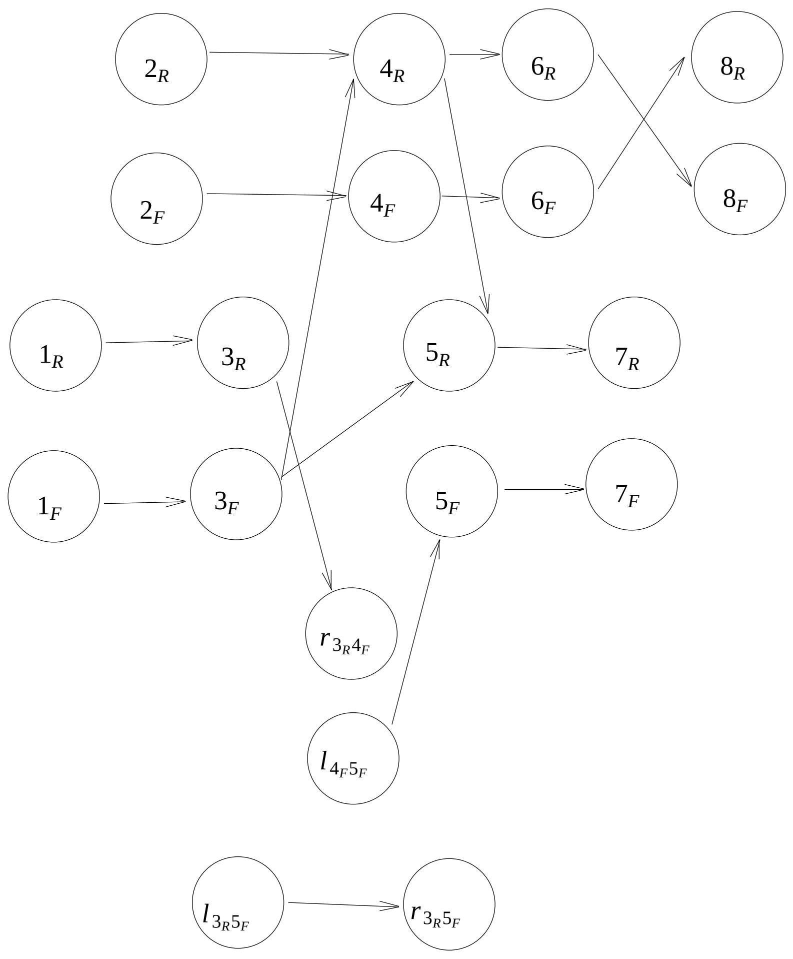


Figure 7: Dag corresponding to network of Figure 3.

Proof: (a) $|cover(N)| \geq |cover(G)|$.

To see this, let $N1 \subset cover(N)$ be the subset of paths that contain edges of type *nn*. From Lemma 4, it follows that each path, $L \varepsilon N1$ contains exactly one *nn* edge. From Lemma 1 and the fact that $cover(N)$ is a network cover, it follows that L covers this *nn* edge. Let $N2$ be the set of *nn* edges in *N*. It follows that $|N1| = |N2|$. Let $T2(N2)$ be the edges in G created by using transformation $T2$ on the edges in $N2$ (actually, $T2(N2)$ consists of single edge paths).

Let L be a path in $N3 = cover(N) - N1$. L may be written as $L_L L_M L_R$. Let L' be the subpath of L that consists of the last edge in L_L (if any), all edges in L_M and the first edge (if any) in L_R. From Lemma 6, we know that L covers exactly the edges in L'. Let $N4$ be the set of paths obtained from $N3$ in this manner. Let $T2(N4)$ be the paths of G obtained by applying transformation $T2$ to each path in $N4$. Since each edge of N is represented at least once in $N2 \cup N4$, it follows that each

dag cover	network path	network cover
$L_1 = l_{3_R 5_F}, r_{3_R 5_F}$	$L_1 = 3_R, 5_F$	$L_1 = 1_R, 3_R, 5_F, 7_F$
$L_2 = l_{4_F 5_F}, 5_F, 7_F$	$L_2 = 4_F, 5_F, 7_F$	$L_2 = 2_F, 4_F, 5_F, 7_F$
$L_3 = 1_R, 3_R, r_{3_R 4_F}$	$L_3 = 1_R, 3_R, 4_F$	$L_3 = 1_R, 3_R, 4_F, 6_F, 8_R$
$L_4 = 1_F, 3_F, 5_R, 7_R$	$L_4 = 1_F, 3_F, 5_R, 7_R$	$L_4 = 1_F, 3_F, 5_R, 7_R$
$L_5 = 1_F, 3_F, 4_R, 5_R, 7_R$	$L_5 = 1_F, 3_F, 4_R, 5_R, 7_R$	$L_5 = 1_F, 3_F, 4_R, 5_R, 7_R$
$L_6 = 2_F, 4_F, 6_F, 8_R$	$L_6 = 2_F, 4_F, 6_F, 8_R$	$L_6 = 2_F, 4_F, 6_F, 8_R$
$L_7 = 2_R, 4_R, 6_R, 8_F$	$L_7 = 2_R, 4_R, 6_R, 8_F$	$L_7 = 2_R, 4_R, 6_R, 8_F$
(a)	(b)	(c)

Figure 8: Covers for examples.

edge of G is covered by the paths in $T2(N2) \cup T2(N4)$. Hence, $|cover(G)| \le |T2(N2)| + |T2(N4)| = |cover(N)|$.

(b) $|cover(N)| \le |cover(G)|$.

Let $G1 \subseteq cover(G)$ be the subset of paths that contain $<l_{ij}, r_{ij}>$ type edges. From the construction of G, it follows that all paths in $G1$ are single edge paths and that $<i,j>$ is an nn edge in N. From Observations 1 and 2, it follows that we can construct a source to sink path of the form $P<i,j>Q$ where P contains yy and ny edges only and Q contains only yy and yn edges. From the definition of the edge classification scheme, it follows that every such $P<i,j>Q$ path covers edge $<i,j>$. Let $G2 = cover(G) - G1$. Let L be a path in $G2$. Replace every occurrence of an l_{ij} (r_{ij}) in this path by $i(j)$. From transformation $T2$, it follows that the resulting path is of the form $L' = [ny,]yy,...,yy[,yn]$ where the ny and yn edges are optional. From Observations 1 and 2, it follows that such a path can be extended to a source to sink path $L'' = PL'Q$ where P (Q) contains only edges of type yy and ny (yy and yn). Hence L'' is in the form $L_L L_M L_R$. From Lemma 7, it follows that L'' covers all edges in L'. In this way, we can obtain from $G2$ a set, $N5$, of paths to cover the yy, yn, ny edges of N. The construction outlined results in $|cover(G)|$ paths that cover all edges in N. Hence $|cover(N)| \le |cover(G)|$.

(a) and (b) together imply that $|cover(N)| = |cover(G)|$. □

Theorem 3: Let N, G, and $cover(G)$ be as in Theorem 2. $cover(N)$ can be obtained from $cover(G)$ by extending each path L' in $cover(G)$ to obtain a source to sink path $L'' = PL'Q$ as in the proof of Theorem 2.

Proof: This follows directly from the proof of part (b) of Theorem 2 and the fact that $|cover(N)| = |cover(G)|$. □

The paths of Figure 8(c) were obtained from those of Figure 8(a) using the construction of Theorem 2 part (b). Since the paths of Figure 8(a) form a dag cover of $G = T2(N)$, it follows that those of Figure 8(c) form a network cover of N.

5 ALGORITHM FOR DAG COVER

A *flow network*, F, is a directed graph which satisfies the following properties:

(1) Each edge $<i,j>$ has a tuple $<L_{ij},U_{ij}>$ associated with it. L_{ij} is the least flow that must go through the edge and U_{ij} is the maximum flow that can go through the edge.

(2) There is a unique source vertex s. This is the only vertex with indegree 0.

(3) There is a unique sink vertex t. This is the only vertex with outdegree 0.

The *flow network* problem is to find a flow through the network such that the flow x_{ij} through edge $<i,j>$ is in the range $[L_{ij},U_{ij}]$ for every edge $<i,j>$ in the network F and the sum of the flows out of the source (this is equal to the flow into the sink) is minimum.

For any dag, G, $cover(G)$ can be found by transforming G into a flow network $F = T3(G)$ and obtaining the minimum flow in F. The transformation $T3$ is described below:

Transformation 3

step1: Create two new vertices s and t.

step2: Add an edge $<s,j>$ from s to every vertex, j, of indegree 0 in G. Let $L_{sj} = 0$, and $U_{sj} = e$ where e is the number of edges in G.

step3: Add an edge $<i,t>$ from every vertex, i, of outdegree 0 in G to vertex t. Let $L_{it} = 0$, and $U_{it} = e$.

step4: Let $L_{ij} = 1$ and $U_{ij} = e$ for all edges in G.

Figure 9 shows $F = T3(G)$ where G is the dag of Figure 7. A minimum flow through F is show in Figure 10(a). The flow through an edge gives the number of paths in $cover(G)$ that include this edge. Figure 10(b) shows a $cover(G)$ that results form this flow.

Theorem 4: $|cover(G)| =$ sum of flows out of the source of $F = T3(G)$ in a minimum flow. Furthermore, $cover(G)$ may be obtained from such a flow by letting each edge be on exactly as many covering paths as the flow through the edge.

Proof: Follows directly from the nature of the transformation and the definition of a minimum flow in F. □

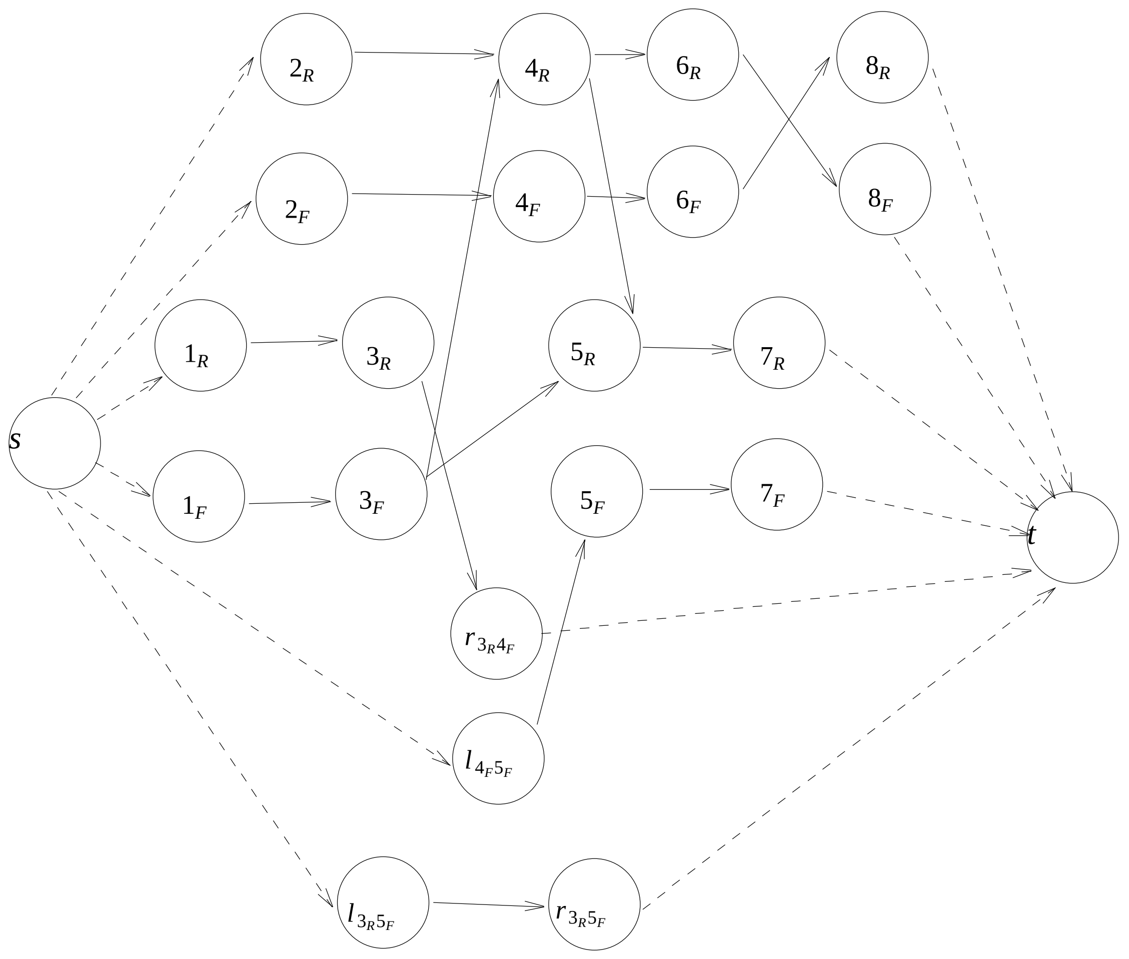
The flow network problem may be solved in $O(m(m+n))$ where n and m are, respectively, the number of vertices and edges in F [7]. The dag cover problem may be solved as below:

step1: construct $F = T3(G)$.

step2: find a minimum flow in F.

step3: construct $cover(G)$ from this flow.

Step 2 dominates the run time as both step 1 and step 3 require time linear in the number of



All solid edges having [1,16] as their range.

All broken edges having [0,16] as their range.

Figure 9: Flow network for dag of Figure 7.

vertices in G and the number of edges in $cover(G)$.

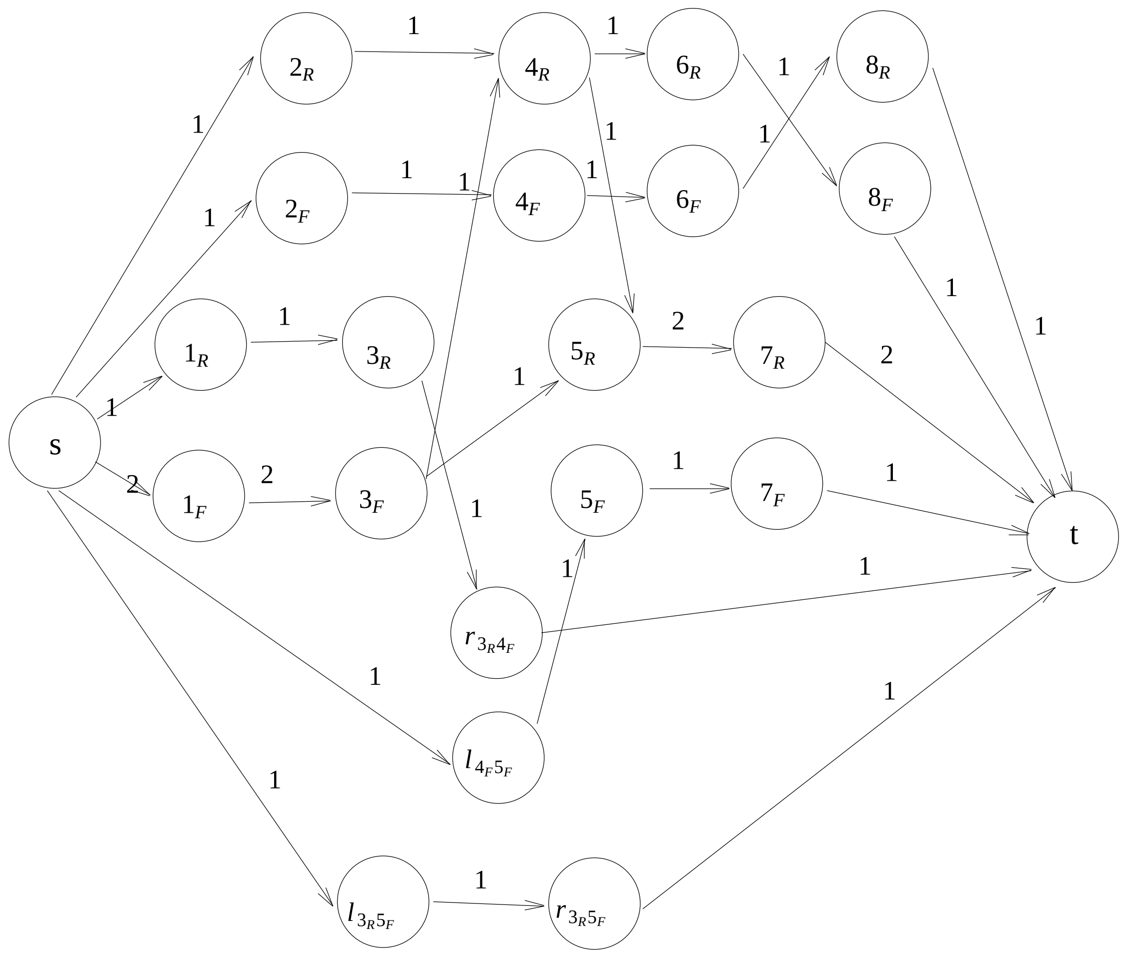
6 EXACT ALGORITHM FOR CIRCUIT COVER

A minimum number of paths covering all edges of C with both rising and falling signals is obtained as follows:

step1: Use Transformation 1 to obtain the network $N = T1(C)$.

step2: Use Transformation 2 to obtain the dag $G = T2(N)$.

step3: Use Transformation 3 to obtain the flow network $F = T3(G)$.



(a)

dag cover

$L_1 = l_{3_R5_F}, r_{3_R5_F}$

$L_2 = l_{4_F5_F}, 5_F, 7_F$

$L_3 = 1_R, 3_R, r_{3_R4_F}$

$L_4 = 1_F, 3_F, 5_R, 7_R$

$L_5 = 1_F, 3_F, 4_R, 6_R, 8_F$

$L_6 = 2_F, 4_F, 6_F, 8_R$

$L_7 = 2_R, 4_R, 5_R, 7_R$

(b)

Figure 10: Minimum feasible flow in flow network of Figure 9.

step4: Find a minimum flow in *F*.

step5: Obtain *cover* (*G*) from this minimum flow.

step6: Obtain *cover* (*N*) from *cover* (*G*).

step7: Obtain *cover* (*G*) from *cover* (*N*).

In actually implementing this algorithm, one could combine $T1$, $T2$, and $T3$ into a single transformation. I.e., F can be obtained directly from C. Furthermore, it is possible to obtain $cover(C)$ directly from the minimum flow of F. All the above steps (except step 4) take time linear in the number of vertices in C and the number of edges in $cover(C)$. Step 4, however, requires $O(m(m+n))$ where n and m are, respectively, the number of vertices and edges in C.

7 FIRST HEURISTIC FOR CIRCUIT COVER - HEURISTIC 1

The number of vertices and edges in practical circuits may be quite large. Hence even though the exact algorithm of the preceding section has a polynomial run time, it may require excessive computer time on real circuits. In this and the next section, we develop linear time heuristics to obtain near minimum circuit covers.

Heuristic 1 uses the network, N, obtained by using Transformation 1 on the given circuit C. It constructs a set of paths that cover the edges in N in a greedy manner. In each iteration, an edge $<i,j>$ is selected to be covered by the next path generated. This edge is selected from a list, L, of edges that haven't been covered by previously constructed paths. In order for the new path to cover $<i,j>$, it must be extended into a source to sink path. This extension is done in such a way as to cover as many additional edges as possible. Let $ExtendToSource(i,j)$ be a procedure that constructs the segment of the path from a source to vertex i. Let $ExtendToSink(i,j)$ be a procedure that constructs the segment of the path from i to a sink vertex.

With respect to our objective of having the new path cover as many additional edges that are not yet covered, we make the following observations.

Observation 3: If $<i,j>$ is of type nn, then from Lemmas 1 and 4, it follows that the new path cannot cover any additional edges. Furthermore, from the proof of Theorem 2, it follows that for the constructed path to cover $<i,j>$, $ExtendToSource$ may use any sequence of yy and ny edges and $ExtendToSink$ may use any sequence of yy and yn edges. During this path extension process, only edge $<i,j>$ is to be marked as covered.

Observation 4: If $<i,j>$ is of type ny, then by constructing a path of type $L_L L_M L_R$ where $<i,j>$ is the last edge of L_L we can cover edge $<i,j>$, all edges in L_M and the first yn edge in L_R (see Lemmas 6 and 7). Hence $ExtendToSource$ simply builds the remainder of L_L using only yy and ny edges. $ExtendToSink$ attempts to use as many not covered yy edges as possible (i.e., make L_M as large as possible), then uses an uncovered yn edge (if necessary). Finally a sink is reached using additional yn and yy edges. The ny edges $<i,j>$, all edges in L_M, and the first edge (if any) in L_R are marked as covered.

Observation 5: If $<i,j>$ is of type yn, then the constructed path can cover only edges $<i,j>$, those in L_M, and the last edge in L_L. The two procedures construct an $L_L L_M L_R$ path to cover as many additional edges as possible. The strategy is similar to that for the case when $<i,j>$ is an ny edge.

Observation 6: If $<i,j>$ is of type *yy* then we attempt to construct an $L_L L_M L_R$ path with $<i,j>$ in L_M that covers as many uncovered edges as possible. Lemma 7 identifies exactly those edges that such a path can cover.

A high level description of Heuristic 1 is given in Figure 11. The edges are selected for covering in topological order. In this order, an edge that may appear before another on a path precedes that edge in the order.

```
HEURISTIC 1
step1:  Construct $N = T1(C)$ using Transformation 1.
step2:  Determine the type ( i.e., yy, nn, yn, ny ) of each edge in $N$.
step3:  Label each edge as not covered.
step4:  while there is an edge that is not covered do
            begin
                let $<i,j>$ be one such edge.
                ExtendToSource $(i,j)$
                ExtendToSink $(i,j)$
            end
step5:  Transform the network paths constructed in Step 4 into circuit paths.
Note1:  Edges not covered are selected in step 4 in topological order.
Note2:  ExtendToSource and ExtendToSink relabel edges covered by the path being constructed
        as covered.
```

Figure 11: Heuristic 1.

Using appropriate data structures, Heuristic 1 can be implemented to have a run time that is linear in the number of edges in $cover(C)$.

8 SECOND HEURISTIC FOR CIRCUIT COVER - HEURISTIC 2

The second heuristic also uses network N obtained from the given circuit C. The basic idea is to start with a levelized network and to iteratively build a cover for paths of maximum weight in N (corresponding to paths of longest delay in circuit C) from source nodes to nodes at level j, until a cover for maximum weight paths to sink nodes is obtained.

It is convenient to introduce two additional edge classifications:

1) type y – —— $<i,j>$ is on a path in $longest(sink(i))$

2) type n – —— $<i,j>$ is not on any path in $longest(sink(i))$

Note that an edge is of type $y-$ ($n-$) if and only if it is of type yy or yn (ny or nn).

The steps of the procedure implementing this heuristic are given below.

Step 1: *Levelize* – Starting with source nodes at level 1 label all nodes such that the level of a node is $(j+1)$ if and only if the maximum of the level number of its predecessors is j.

Step 2: *Mark* – Mark each edge in N as $y-$ or $n-$ according to the definitions given earlier.

Step 3: Let $j=1$. Let $pp(i)$ be the currently chosen, by the heuristic being used, (partial) paths into node i from source nodes.

Step 4: *Extend_Path* – Extend paths in $pp(i)$ into each non-sink node i at level j such that the following conditions are met:

(i) each path into a non-sink node, say i, is extended by at least one type $y-$ edge that is outbound from node i,

(ii) each type $y-$ edge that is out bound from node i extends at least one $longest(source(i))$ path in $pp(i)$ and

(iii) each type $n-$ edge that is out bound from node i extends exactly one $longest(source(i))$ path in $pp(i)$.

Step 5: Let $j=j+1$. Derive $pp(i)$ for each node i in level j by collecting appropriate paths created in Step 4.

Step 6: If j is less than maximum level number in N then go to Step 4.

Step 7: The cover for network N is the aggregate of paths collected in Step 6 for sink nodes.

The main feature of this heuristic is that it increases the number of paths being selected only when it encounters an out bound edge at a non-sink node i which cannot be attached to an already selected partial path into node i. The increase in the number of selected paths occurs only in attempting to satisfy conditions (ii) and (iii) of Step 4.

Given a network N the algorithm given above can be executed on N from source to sink nodes (called forward direction) and can also be executed from original sink nodes of N to original source nodes of N by reversing the direction of edges (this is called backward direction). By appropriate choice of data structures the algorithm stated above executes in time that is linear in the number of edges in $cover(C)$.

9 EXPERIMENTAL RESULTS

The exact and heuristic algorithms for the circuit cover problem have been implemented in C and run on 10 circuits called ISCAS circuits [6]. The characteristics of these circuits are provided in Figure 12. Run times are provided in Figure 13. The run times are in seconds. The computer used is an Apollo DN3000. Two versions of Heuristic 1 were programmed. In heur1(f) the vertices were processed in topological order. In heur1(b), they were processed in the reverse of this order. As can be seen, changing the processing order did not affect its run time materially. Heur2, on the other hand, generally runs significantly faster when the circuit is processed from sinks to sources (b) rather than from sources to sinks(f). Heuristics 1(f) and 1(b) obtained optimal covers for all 10 circuits. Heuristic 2(b) failed to obtain an optimal cover only for circuit 9. For this circuit, the cover obtained had 3875 paths rather than 3797. Heuristic 2(f) failed to obtain an optimal cover for circuits 9 and 10. The number of paths in the covers obtained for these circuits was 3875 and 5432 respectively. All four heuristics have a comparable run time. The exact algorithm is much slower.

10 CONCLUSIONS

We have developed a polynomial time algorithm to find a minimum cardinality path set that can be used to verify the correct operation of a digital circuit. In addition, two efficient heuristics have been developed. Experimental results indicate that these heuristics obtain near optimal solutions (in fact, heuristic1 obtained optimal solutions in all test cases) while using only a fraction of the computer time required by the minimum cardinality algorithm.

Even though we have assumed that the combinational logic circuit under consideration is constructed from AND, OR, NAND, NOR and NOT gates, it is clear that circuits containing other types of gates can be accommodated by using an appropriate circuit model for such gates.

11 REFERENCES

1 G.L Smith, "Model for Delay Faults Based Upon Paths," *Proc. 1985 Int'l. Test Cnf.*, Nov. 1985, pp. 342-349.

2 Y.K. Malaiya and R. Narayanaswamy, "Testing for Timing Faults in Synchronous Sequential Integrated Circuits," *Proc. 1983 Int'l. Test Conf.*, Oct. 1983, pp. 560-571.

3 K.D. Wagner, "Delay Testing of Digital Circuits Using Pseudorandom Input Sequences," Center for Reliable Computing Report 85-12, revised March 1986, Stanford University.

4 J. Savir and W.H. Mcanney, "Random Pattern Testability of Delay Faults," *Proc. 1986 Int'l. Test Cnf.*, Sept. 1986, pp. 263-273.

5 C.J. Lin and S.M. Reddy, "On Delay Fault Testing in Logic Circuits," *IEEE Trans. CAD,* Sept. 1987, pp. 694-703

6 F. Brglez and H. Fujiwara, "Neutral Netlist of Ten Combinational Benchmark Circuits and a Target Translator in FORTRAN," *Proc. IEEE Int. Symp. Circuits & Systems*, June 1985

7 E. Lawler, *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston. 1976

8 R.B. Hitchcock, G.L. Smith and D.D. Cheng, "Timing Analysis of Computer Hardware," *IBM J. Research & Development*, vol. 26, No. 1, Jan. 1982, pp. 100-108.

#	circuit	# vertices	# edges	size of optimal cover
1	c432	250	426	402
2	c499	555	928	808
3	c880	443	729	729
4	c1355	587	1064	848
5	c1908	913	1498	1284
6	c2670	1426	2076	1871
7	c3540	1719	2939	2563
8	c5315	2485	4386	4353
9	c6288	2448	4800	3797
10	c7552	3719	6144	5431

Figure 12: Circuit characteristics.

#	heur 1(f)	1(b)	2(f)	2(b)	exact
1	1.383	1.350	1.400	1.350	14.133
2	2.767	2.717	3.017	3.000	69.983
3	2.000	1.983	2.567	2.050	59.067
4	2.983	2.983	3.350	3.383	45.767
5	4.733	4.633	5.433	6.017	172.083
6	7.183	6.983	8.200	6.283	390.717
7	10.400	10.217	15.633	9.350	584.867
8	16.033	15.883	18.717	13.233	1846.417
9	23.250	23.050	50.833	30.700	1178.017
10	24.433	24.400	24.117	20.600	3127.433

Figure 13: Algorithm run times (in seconds)