Minimal Cutset Enumeration and Network Reliability Evaluation by Recursive Merge and BDD

Hung-Yau Lin Sy-Yen Kuo Fu-Min Yeh
Department of Electrical Engineering *Chung-Shan Institute of Science*
National Taiwan University, Taipei, Taiwan *and Technology, Taoyuan, Taiwan*
hylin@lion.ee.ntu.edu.tw *sykuo@cc.ee.ntu.edu.tw* *fmyeh@tpts5.seed.net.tw*

Abstract

One of the key tasks in network reliability evaluation is to enumerate all the paths or minimal cutsets of a network. Then the reliability can be calculated from the disjoint form of these terms. Enumerating all the minimal cutsets may be a feasible way to evaluate the reliability of a network if the number of paths is too huge to enumerate practically. One example of this kind of networks is the 2x100 lattice network. Many algorithms have been proposed to enumerate the minimal cutsets of a graph. Most of them require advanced mathematics or can only be applied to either one of the two broad categories, directed and undirected graphs. This paper presents a simple and systematic recursive algorithm that guarantees the generated cutsets are minimal and the same logic can be applied to both directed and undirected graphs with ease. This algorithm is so simple to implement and efficient that it can also be used to check the correctness of the cutsets generated by other algorithms. This algorithm can also be combined with OBDD (Ordered binary decision diagram) to calculate the reliability of a network. Experimental results show that: (1) the running time of enumerating all cutsets versus the graph density is linear for a given number of nodes and (2) it takes 96.71 seconds to evaluate the network reliability of a 2x100 lattice network which has 2^{99} paths.

1. Introduction

One of the key tasks in network reliability evaluation is to enumerate all the paths or the minimal cutsets of a network. Then the reliability can be calculated from the disjoint form of these terms [1-2]. Some networks have such a huge number of paths that it is simply impractical to enumerate all these paths. If the number of minimal cutsets is far less than the number of paths, then it may be a feasible way to evaluate the reliability through cutsets. A 2x100 lattice network is an example of such networks. It has 2^{99} paths but contains only 10,000 minimal cutsets.

One other method evaluates the reliability by applying the edge expansion diagram and OBDD (Ordered binary decision diagram) without enumerating all the paths or cutsets [3-4]. Though this method can evaluate network reliabilities very efficiently, it does not produce the pathset (set of all paths) or cutsets of a network. The pathset or cutsets may be useful in some applications such as network management or network flow control and calculation. Our algorithm can generate all minimal cutsets and evaluate the reliability. It is even faster than the method in [4] for the complete network with 10 nodes.

Many algorithms have been proposed to generate minimal cutsets for directed or undirected graphs [5-13]. Some of the algorithms need special preprocessing on the graphs or can only be applied to a special kind of graphs [9, 12, 13]. Some algorithms require advanced mathematics [5] and some others can only be applied to undirected networks. One other approach obtains the minimal cutsets by inverting minimal pathset [10] but it may be impractical to generate the pathset of a graph.

Tsukiyama [7] presented methods and mathematical background to enumerate all the minimal cutsets (also called *s-t cutset* in their paper) of a graph. The way of choosing vertices can be made even clearer and more systematic. This paper presents a simple and systematic algorithm that guarantees the generated cutsets are minimal and the same logic can be applied to both directed and undirected graphs with ease. This algorithm is very simple to implement and is efficient that it can also be used to check the correctness of other algorithms [6, 14]. This algorithm can also be combined with BDD

Acknowledgement: This research was supported by the National Science Council, Taiwan, R. O. C., grant NSC 91-2213-E-002-042.

to calculate the reliability of a network. Unfortunately, enumerating all the cutsets is a well-known NP-hard problem [15] and our experiments demonstrate this exponential growth rate of the computation time. Experimental results also show that: (1) the running time versus the graph density is linear for a given number of nodes and (2) it takes 96.71 seconds to evaluate the network reliability of a 2x100 lattice network which has 2^{99} paths.

2. Preliminaries

Notations:

V, E	set of vertices and edges respectively.
G	a graph, $G = [V, E]$.
(u, v)	an undirected edge connecting vertices u and v. u and $v \in V$.
$<u, v>$	a directed edge from u to v. u and $v \in V$.
s, t	source node and sink node respectively.
S, T	disjoint subsets of V such that $s \in S$ and $t \in T$.
SS	(source set) set of vertices such that $s \in SS$ and SS is connected.
$G*n$	node n is merged into SS of G by deleting any edge connecting n and SS.

A graph is connected if there is a path between each pair of vertices. A graph $G' = [V', E']$ is a subgraph of G if $V' \subset V$ and $E' \subset E$. A maximal connected subgraph of a graph G is called a *connected component* of G. For any subset X of V, a graph consisting of a set of vertices X and a set of edges $E[X] \equiv \{e = (u, v) \in E \mid u, v \in X\}$ is designated as a *section subgraph* of G and is denoted by $G[X] = [X, E[X]]$. A node v is said to be *adjacent* to $G[X]$ if there exists an edge $<u, v>$ or (u, v) such that $u \in X$ and $v \notin X$. In other words, node v is adjacent to X if there is an edge leading to node v from X. Let $s \in S$ and $t \in \overline{S} \equiv V - S$ and let the set of edges connecting S and $\overline{S}$ represent a cutset $C = w(S, \overline{S}) \equiv \{(u, v) \in E \mid u \in S, v \in \overline{S}\}$. If C does not contain any other cutset, then it is referred to as a *minimal cutset*.

According to Lemma 4 in [7], if we can find two connected components S and $\overline{S}$, $s \in S$ and $t \in \overline{S}$, such that any node adjacent to S is in $\overline{S}$, then there exists a single minimal cutset $C = w(S, \overline{S})$. Lemma 3 in [7] suggests that if $\overline{S}$ is not connected, then any other node not included in a connected component W, which contains t, can be merged into S to form a new S. For detailed proof, please refer to [7]. Our algorithm is based on these two Lemmas. To prove that our algorithm does indeed enumerate all the minimal cutsets, we need to show that: (1) our algorithm does not produce isolated nodes in the intermediate subgraph and (2) any cutset generated by the algorithm in [7] can also be produced by our recursive merge algorithm. The proof is given in section 4 after the algorithm is illustrated.

3. The algorithm for undirected graphs

For any connected graph G, it is obvious that deleting all the edges emitting from s prevents s from arriving t. If one of these edges is not deleted, then s has a way out and can get to t since the rest of the graph is connected. Therefore, this is a minimal cutset of G. The basic idea is to generalize the source node into the source set. Then the minimal cutset is composed of all the edges emitting from the source set.

The algorithm is shown in the pseudo-code form in Figure 1 and it is invoked by initializing G to the original graph, SS to empty, and n to s. In the algorithm, we merge the nodes adjacent to the source set one by one and absorb (merge) redundant nodes of G into SS. This can ensure that the set of all emitting edges from a particular SS is a minimal cutset. In the conventional sense, a node is called a redundant node if a node n other than s or t is connected to only one other node. In this paper, a node is called a *redundant node* if it is adjacent to SS and has no way to get to t without going through any node in SS. If an SS is found in the hash table, it has been evaluated in the recursive process and nothing needs to be done. If an SS is not found in the hash table, then add it to the hash table and then output a cutset. A cutset is the set of all edges emitting from SS. Any node adjacent to SS is then merged to the SS one by one to form a new SS and call the recursive function. The SS can be thought of as being a single node that represents the original s. A node is merged into SS by deleting any edge linking this node and any node in SS while maintaining all other edges.

```
RecursiveMerge( G, SS, n )
{
    if ( n == t )    return;
    G = G*n;    SS = SS + n;
    Recursively absorb (merge) redundant nodes of G into SS;
    if ( SS is found in the hash table )    return;
    else    {add SS to the hash table;}
    Output a cutset of SS;
    for each node n_i adjacent to SS
    {
        RecursiveMerge( G, SS, n_i );
    }
}
```

Figure 1. Minimal cutsets enumeration algorithm.

Figure 2 illustrates this algorithm and the graph contains 5 nodes and 6 edges. Node 1 and 5 are s and t

respectively. The RecursiveMerge algorithm begins by setting *SS* to empty and *n* to 1. The rectangle G1 in Figure 2 outputs a cutset {e1, e2}. There are two nodes, 2 and 3, adjacent to *SS* in G1 and *SS* is merged with them to produce G2 and G3. G2 then merges with node 3 into G4 to have a *SS* containing {1, 2, 3} and output a cutset {e3, e4, e5}. The *SS* in G2 can also merge with node 4 to have *SS* = {1, 2, 4}. As we can see in the figure that node 3 will become a redundant node and have to be absorbed into *SS*. It will create G6 by absorbing node 3 into *SS*. Of course, if G6 has not been found in the hash table, it will produce a cutset and call the recursive merge function as a normal sub-graph does. If G3 merges with node 2, its *SS* will contain {1, 2, 3} and it is found in the hash table. Therefore, no cutset shall be output and no recursive processing is needed. There are 6 rectangles G1 to G6 in Figure 2 and this means we have 6 cutsets for this graph.

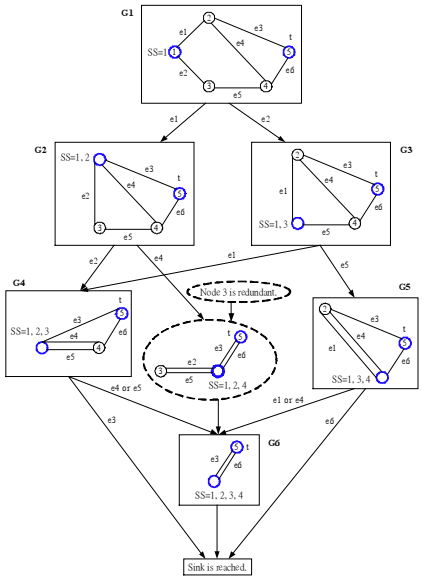

Figure 2. Example of the recursive merge algorithm.

4. The algorithm for directed graphs

The above algorithm can also be applied to directed graphs without modification. But care must be taken as to what nodes can be merged to and what nodes can be absorbed into *SS*. In the directed graph, a node *v* can be merged into *SS* if there is a node $u \in SS$ such that $<u, v>$ is an edge connecting from *u* to *v*. Similarly, a node *n* can be absorbed into *SS* if there is a node $u \in SS$ such that $<u, n>$ is an edge and *n* have no way to get to *t* without going through any node in *SS*. Consider an intermediate sub-graph of a directed graph shown in Figure 3. If e4 does not exist, then both node *x* and *y* are redundant nodes and shall be absorbed into *SS*. If e4 does exist, then only node *x* is redundant and absorbed into *SS*. Therefore, if a node *n* is not a redundant node, any other node reachable from *n* is not necessarily a non-redundant node. But if a node *n* is a redundant node, then any node $r \notin SS$ reachable from *n* is also a redundant node. Otherwise, *n* will have a path to *t* through *r*.

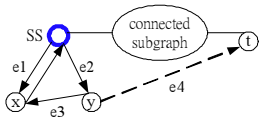

Figure 3. An intermediate sub-graph (directed).

For an undirected graph, the performance of the recursive merge algorithm can be improved if we can absorb a group of redundant nodes into *SS*. Consider an intermediate sub-graph of an undirected graph shown in Figure 4. If edge e4 does not exist, node *x* and *y* are both redundant nodes and shall be absorbed into *SS*. Obviously, if e4 exists, both node *x* and *y* are not redundant nodes and shall not be absorbed into *SS*. In other words, if a node *n* is a redundant node in an undirected graph, any node $r \notin SS$ reachable from *n* is also a redundant node. Therefore, a group of nodes can be absorbed into *SS* in the undirected graph and it speeds up the enumeration process. In some graphs, we have observed up to 3 times speedup compared to the approach that absorbs one node at a time. Other sophisticated methods or data structures can also be devised to improve the performance in the hashing mechanism or in finding the redundant nodes of an intermediate *SS*.

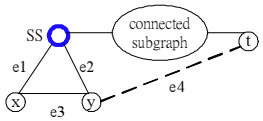

Figure 4. An intermediate sub-graph (undirected).

The following two Lemmas and the Lemmas in [7] can prove that our recursive merge algorithm does indeed generate complete enumerations of minimal cutsets. The Lemmas in [7] are not described again in this paper.

Lemma 1: For the input of a connected graph, the recursive merge algorithm does not produce isolated nodes in any intermediate subgraph.

[Proof.] In the recursive merge algorithm, merge (or absorption) is the only process that will delete edges. According to the algorithm, a set of nodes *SS* can be merged together if they are connected. Besides, for any

node $u \in V - SS$ and $v \in V$, any edge connecting u and v, whether it is (u, v), $<u, v>$, or $<v, u>$, is not deleted. In other words, they are connected to some other nodes. Thus, the recursive merge algorithm does not produce isolated nodes in any intermediate subgraph.

Lemma 2: Any cutset generated by the algorithm in [7] is also produced by our recursive merge algorithm.

[Proof.] The algorithm in [7] partitions the graph into two connected components S and $\overline{S}$, $s \in S$ and $t \in \overline{S}$, such that any node not in S is in $\overline{S}$. Then, the edges connecting S and $\overline{S}$ is a minimal cutset. In the recursive merge algorithm, SS is connected since only connected nodes can be merged together. We want to show t is in $\overline{SS}$ and $\overline{SS}$ is also connected. In the algorithm, recursion terminates if t is reached. Besides, t is not a redundant node by definition. Therefore, t is in $\overline{SS}$. By definition, a node is called a *redundant node* if it is adjacent to SS and has no way to get to t without going through any node in SS. Consider a redundant node v not adjacent to SS. There must be a redundant node u adjacent to SS and can reach v. Each node along the path from u to v is also redundant because if they are not redundant, then u will not be redundant. The nodes along this path can be absorbed (merged) into SS. Thus, any redundant node can be absorbed into SS. In other words, any node $n \in \overline{SS}$ has a way to t. Thus, $\overline{SS}$ is connected.

5. Reliability Evaluation with BDD

The BDD [3] is based on a decomposition of a Boolean function called the Shannon expansion. A function f can be decomposed in terms of a variable z as:

$$f = z \cdot f_{z=1} + \overline{z} \cdot f_{z=0}$$

In the equation above, $f_{z=1}$ is called the *positive cofactor* of f with respect to z, i.e., the result of evaluating f with $z = 1$. Similarly, $f_{z=0}$ is the *negative cofactor* of f. The node z and its descendants in an BDD represent a Boolean function f; one outgoing edge of z is directed to the subgraph representing $f_{z=1}$, and the other is to the subgraph representing $f_{z=0}$. In a BDD, A dashed line is for taking the value 0 and a solid line for value 1. Following a path from the root to a terminal node, we can simply take successive cofactors of a function until a terminal node is reached.

BDD has a useful property, which guarantees that all paths from the root to a terminal node are mutually disjoint. Thus, the recursive merge algorithm in Figure 1 can be modified to evaluate the unavailability of a graph by using BDD. This algorithm is shown in Figure 5. The gBdd in the RM_BDD is a global BDD variable initialized to bdd_Zero. Assume $P(z)$ is the success probability of variable z. After all the cutsets has been generated, gBdd can be used to evaluate the unavailability by:

$$\Pr(gBdd) = (1 - P(z_i)) \times \Pr(gBdd_{z_i=1}) + P(z_i) \times \Pr(gBdd_{z_i=0})$$

The summation of unavailability and availability (reliability) should be 1. Therefore, we can get the reliability of a graph by subtracting unavailability from 1. The BDD and the unavailability of Figure 2 are shown in Figure 6. The success probability of each edge is assumed to be 0.9. As we can see from Figure 6, the reliability of Figure 2 is 1 – 0.031078 = 0.968922.

```
RM_BDD(G, SS, n )
{
    bdd temp_bdd;
    temp_bdd = bdd_One;
    if ( n == t )   return;
    G = G*n;   SS = SS + n;
    Absorb (merge) redundant nodes of G into SS;
    if ( SS is found in the hash table )   return;
    else   {add SS to the hash table;}
    Output a cutset C of SS;
    for each edge e_i in the cutset C
        temp_bdd = bdd_And( e_i, temp_bdd );
    gBdd = bdd_Or( gBdd, temp_bdd );
    release temp_bdd;
    for each node n_i adjacent to SS
    {
        RM_BDD( G, SS, n_i );
    }
}
```

Figure 5. BDD construction with recursive merge.

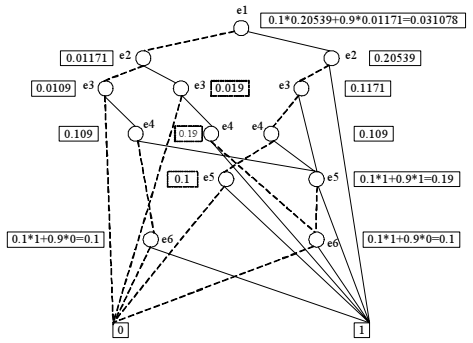

Figure 6. Unavailability and BDD of Figure 2.

6. Experimental Results

There exist various methods to derive the SDP (sum of disjoint products) terms from the pathset or cutsets of a network. Then the reliability can be obtained by evaluating the SDP terms. Soh [2] showed that different pre-processing methods could produce different numbers of SDP terms and thus have different computation time. Unfortunately, enumerating all paths or cutsets for a large network is simply impractical. Kuo [4] has presented a very efficient method to tackle this problem. Their method does not enumerate all the paths or cutsets. Instead, they use the edge expansion diagram and the binary decision diagram (BDD). The reliability is then evaluated by traversing the BDD. In some applications, such as network management or network flow control and calculation, the set of paths or cutsets may be very useful.

We apply the DeMorgan's law to the *Path_Function_Construct* algorithm in [4] and derive from it a cut-based algorithm using edge expansion diagram. We also implement that cut-based algorithm and name it *Cut_EED* in this paper for identification and comparison. Twelve benchmark networks from [2, 4] are tested and they are shown in Figure 7.

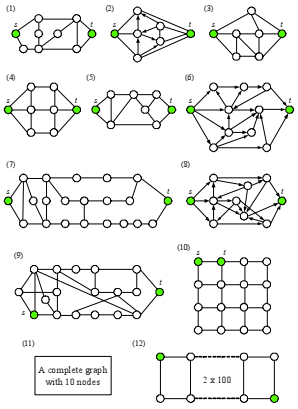

Figure 7. Twelve benchmark networks.

Both *Cut_EED* and *RM_BDD* are written in C++ and compiled with gcc 3.2 on a Linux system. Our hardware system has a Pentium-III 1GHz CPU and 512 MB of memory. The experimental results for these 12 benchmark networks are shown in Table 1. The unit of time is in seconds. Note that the running time in [2] does not include the time to enumerate all the paths or cutsets while ours includes the time to enumerate all cutsets.

The edge expansion diagram used by Cut_EED can delete more edges at each step than the recursive merge algorithm RM_BDD. Besides, the edge expansion diagram produces much fewer sub-problems than the recursive merge algorithm. Therefore, algorithms based on the edge expansion diagram are in general faster than RM_BDD. But this is not true for network 11. The edge expansion diagram produces a lot of sub-problems just as RM_BDD does. RM_BDD stops recursive function call if an *SS* is found in the hash table and it applies BDD operations (AND or OR) on each cutset. Unlike RM_BDD, Cut_EED also applies BDD operations to the isomorphic sub-graphs. Other than these facts, network 11 also has much less cuts than paths. These may be the reason why RM_BDD is faster than Cut_EED for network 11.

Table 1. Comparisons of benchmark networks.

Network	# of paths	# of cuts	Reliability	Lex [2]	Cut_EED	RM_BDD
1	13	28	0.964855	0	0	0
2	14	18	0.996664	0	0	0
3	25	20	0.997494	0	0	0
4	29	29	0.996217	0.1	0	0
5	24	19	0.975116	0	0	0
6	18	110	0.994076	0.1	0	0.03
7	44	528	0.904577	0.3	0	0.14
8	64	78	0.997506	0.1	0	0.02
9	281	1,300	0.985928	3.8	0.01	0.71
10	98	105	0.987831	-	0.01	0.03
11	109601	256	1.0	-	2.17	1.36
12	2^{99}	10,000	0.304317	-	0.24	96.71

For some networks, it is simply impractical to enumerate all paths since there are a huge number of paths. Consider network 12 in Figure 7. Though it has 2^{99} paths, it contains only 10,000 cutsets. Though Cut_EED is faster than RM_BDD, applications other than reliability evaluation may need pathsets or cutsets. Enumerating their cutsets seems to be a feasible alternative way for such networks. RM_BDD can calculate the reliability of network 12 within reasonable time.

We also run the recursive merge algorithm on some randomly generated graphs and measure their time to enumerate all the cutsets. Figure 8 shows the result of running time versus density. The number of nodes ranges from 16 to 20 and the density is from 0.4 to 1.0 at a step size of 0.1. The graph density is defined as $2m/(n(n\text{-}1))$ where m and n are the number of edges and the number of nodes respectively. From Figure 8, we can see that for a particular node size, the growth rate of the processing time is linear to the density of a graph. Table 2 shows the detailed results for 19-node and 20-node networks.

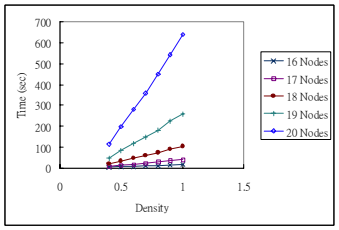

Figure 8. Processing time versus graph density.

Unfortunately, hidden under the linear relationship between time and the density is the nature of exponential growth rate. This exponential growth rate is demonstrated in Figure 9 where we plot the time to enumerate all the cutsets versus the number of nodes. The number of cutsets increases with the number of nodes so rapidly that it is impossible to enumerate all the cutsets of a large graph.

Finally, we also verify the network in [14] and confirm that our algorithm also generates 214 cutsets.

Table 2. Detailed experimental results.

Density	20 nodes		19 nodes	
	# of cuts	Time (sec)	# of cuts	Time (sec)
0.4	186233	112.86	99528	45.94
0.5	243036	197.94	121940	83.22
0.6	258473	282.16	128501	116.68
0.7	261266	357.33	130614	148.96
0.8	261876	449.48	130969	182.13
0.9	262129	541.39	131044	225.76
1.0	262144	640.78	131072	259.41

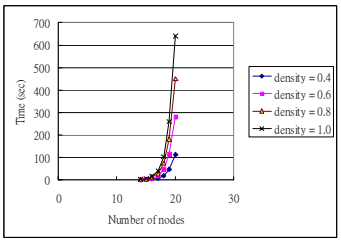

Figure 9. Exponential growth nature of cutset enumeration.

7. Conclusions

This paper presents a simple recursive algorithm that employs the idea of generalizing the redundant node and the source node to the source set. The minimal cutset is composed of all the edges emitting from the source set. The algorithm can also be combined with BDD to evaluate network reliabilities. The same logic of the algorithm can be applied to both directed and undirected graphs with ease. This algorithm is very simple to implement that it can also be used to check the correctness of other algorithms. Unfortunately, enumerating all the cutsets is a well-known NP-hard problem and our experiments demonstrated this exponential growth rate of the computation time. Experimental results also show that: (1) the running time versus the graph density is linear for a given number of nodes and (2) it takes only 96.71 seconds to evaluate the network reliability of a 2x100 lattice network which has 2^{99} paths.

References

[1] S. Rai and K. K Aggarwal, "An Efficient Method for Reliability Evaluation of a General Network," *IEEE Trans. Reliability*, vol. 27, pp. 206-211, Aug 1978.

[2] S. Soh and S. Rai, "Experimental Results on Preprocessing of Path/Cut Terms in Sum of Disjoint Products Techniques," *IEEE Trans. Reliability*, vol. R-42, no. 1, pp. 24-33, Mar 1993.

[3] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. C-35, no. 8, pp. 677-791, Aug 1986.

[4] Sy-Yen Kuo, Shyue-Kung Lu, and Fu-Min Yeh, "Determining Terminal-Pair Reliability Based on Edge Expansion Diagrams Using OBDD," *IEEE Trans. Reliability*, vol. 48, pp. 234-246, Sep 1999.

[5] Alberto Martelli, "A Gaussian Elimination Algorithm for the Enumeration of Cut Sets in a Graph," *Journal of A.C.M.*, vol. 23, pp. 58-73, Jan 1976.

[6] S. Arunkumar and S. H. Lee, "Enumeration of All Minimal Cut-Sets for a Node Pair in a Graph," *IEEE Trans. Reliability*, vol. 28, pp. 51-55, Apr 1979.

[7] S. Tsukiyama, I. Shirakawa, and H. Ozaki, "An Algorithm to Enumerate All Cutsets of a Graph in Linear Time Per Cutset," *Journal of A.C.M.*, vol. 27, no. 4, pp. 619-632, Oct 1980.

[8] U. Abel and R. Bicker, "Determination of All Minimal Cut-Sets between a Vertex Pair in an Undirected Graph," *IEEE Trans. Reliability*, vol. R-31, no. 2, pp. 167-171, Jun 1982.

[9] C. S. Sung and B. K. Yoo, "Simple Enumeration of Minimal Cutsets Separating 2 Vertices in a Class of Undirected Planar Graphs," *IEEE Trans. Reliability*, vol. R-41, no. 1, pp. 63-71, Mar 1992.

[10] D. R. Shier and D. E. Whited, "Algorithms for Generating Minimal Cutsets by Inversion," *IEEE Trans. Reliability*, vol. 34, pp. 314-319, Oct 1985.

[11] D. R. Shier and D. E. Whited, "Iterative Algorithms for Generating Minimal Cutsets in Directed Graphs," *Networks*, vol. 16, pp. 133-147, 1986.

[12] S. H. Ahmad, "Simple Enumeration of Minimal Cutsets of Acyclic Directed Graph," *IEEE Trans. Reliability*, vol. R-37, no. 5, pp. 484-487, Dec 1988.

[13] L. Yan, and H. A. Taha, "A Recursive Approach for Enumerating Minimal Cutsets in a Network," *IEEE Trans. Reliability*, vol. 43, pp. 383-388, Sep 1994.

[14] V. Sankar, V. C. Prasad, and K. S. Parakasa Rao, "Comment on: Enumeration of All Minimal Cutsets for a Node Pair in a Graph," *IEEE Trans. Reliability*, vol. R-42, no. 1, pp. 44-45, Mar 1993.

[15] M. O. Ball, "Computational Complexity of Network Reliability Analysis: An overview," *IEEE Trans. Reliability*, vol. 35, pp. 230-239, Aug 1986.