A NEW ALGORITHM FOR NETWORK RELIABILITY COMPUTATION*

A. Grnarov1, L. Kleinrock2 and M. Gerla3

1Electrical Engineering Department
University of Skopje, Yugoslavia

2UCLA Computer Science Department
Los Angeles, California 90024

I. Introduction

The analysis of reliability networks has been the subject of considerable research. The terminal reliability algorithms available in the literature are based on state enumeration,1-4 factoring,5 reduction to series-parallel network,6,7 path enumeration,8-20 cut-set enumeration,21,22 and case analysis23. Recently Satyanarayana and Prabhakar24 proposed an efficient algorithm based on the acyclic subgraphs of the given probabilistic graph.

The most efficient path enumeration algorithms use reduction to mutually exclusive events by Boolean algebra. The drawbacks of these algorithms, however, are the fact that they generate a large number of terms,16,20 they can efficiently handle systems of only moderate size (say, fewer than 20 paths or cut-sets between the input-output node pair),11 and they do not permit an efficient determination of the resulting function when the number of elements in the network is large.15

In this paper a new algorithm for terminal reliability computation of a general network is presented. The algorithm belongs to the path enumeration algorithms (which use Boolean algebra) and is based on performing the here defined S-operation (modified #-operation)25 on the set of all simple paths. A simple path is represented by a particular binary string (path identifier); thus, only logical operations need to be performed. The proposed algorithm is more efficient than existing algorithms since it does not suffer from the previously mentioned drawbacks. It can be applied to both oriented and non-oriented networks and easily modified to produce symbolic expression for terminal reliability.

The algorithm was coded in FORTRAN IV and run on a DEC-10 timesharing computer system. Execution times confirm the advantages of the proposed algorithm over existing algorithms.

II. Derivation of the Algorithm

For a network consisting of N nodes and E links, we consider the set of paths between a given source s and a given destination t. Each path in the set is identified by a path identifier defined as follows:

*This research was supported in part by the Advanced Research Projects Agency of the Department of Defense under contract MDA 903-77-0272. 

Definition 1: The path identifier \( IP_k \) for the path \( \pi_k \) is defined as a string of \( n \) binary variables

\[
IP_k = x_1x_2 \cdots x_n
\]

where

\[
x_i = 1 \quad \text{if the}\ i\ \text{th element of the network (node or link) is included in the path}\ \pi_k
\]

and \( x_i = 0 \) otherwise

\( n = \text{number of network elements that can fail, i.e.:} \)

\( n = N \quad \text{in the case of perfect links and imperfect nodes} \)

\( n = E \quad \text{in the case of perfect nodes and imperfect links} \)

\( n = N + E \quad \text{in the case of imperfect links and imperfect nodes} \)

According to Definition 1, it can be seen that a path identifier has the form of the cube in Boolean algebra. Hence, by applying the #-operation ("sharp" operation) between two path identifiers, denoted by \( IP_k \# IP_j \), we obtain the set of all subcubes (events) of \( IP_k \) not included in \( IP_j \). Unfortunately, the generated subcubes are not mutually disjoint. The #-operation should be repeated on the set of subcubes again and again until a set of disjoint subcubes is obtained.

In the case when the paths \( \pi_j, \quad j = 1, 2, \ldots, m-1, \) have been examined, the determination of the set of subcubes of the path \( \pi_m \), which are not included in the previous paths, can be performed as

\[
S_m = (\ldots (IP_m \# IP_p) \# IP_p) \# \ldots \# IP_p) \# \ldots \# IP_{m-1}
\]

Since the #-operation is performed using the largest possible cubes (path identifiers), the generation of \( S_m \) is faster than with Fratta's10 and Rai's13 methods, and also there is no need for the storing of the new terms. The drawbacks of performing (1) are the generation of the repeated subcubes and the need for the repeated application of the #-operation between the subcubes in \( S_m \). To avoid these drawbacks a new S-operation, called "exclusive sharp" operation, will be introduced later in Definition 4.

Before the presentation of Definition 4 it is useful to make the following consideration. Since the path identifiers contain only symbols 1 and x, the #-operation by definition (see Table 1 below) can produce only 0 as a new symbol.
The symbols 0 generated in step \( j \) are designated as \( 0^j \). If in the cube generated by \#-operation there is only one \( 0^j \), we call this a unique 0. Now we quote the definition of the coordinate \#-operation between two cubes as given in Miller et al\(^{(25)}\) (page 173), and we also present the definitions of the \(@_j\)-operation and the \(S\)-operation.

**Definition 2.** The coordinate \#-operation is defined as given in Table 1.

<table>
<thead>
<tr>
<th>( a_i )</th>
<th>( b_i )</th>
<th>( 0 )</th>
<th>( 1 )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>z</td>
<td>y</td>
<td>z</td>
<td>x</td>
</tr>
<tr>
<td>1</td>
<td>y</td>
<td>z</td>
<td>z</td>
<td>1</td>
</tr>
</tbody>
</table>

**Definition 3.** The \(@_j\)-operation between two cubes, say \( C' = a_1 a_2 \cdots a_n \), and \( C'' = b_1 b_2 \cdots b_n \), is defined as

\[
C' \circ_j C'' = \begin{cases} 
C' & \text{if } a_i \# b_i = y \text{ for any unique 0 or } v \\
C' \cup C'' & \text{if } a_i \# b_i = y \text{ for all } a_i = 0^j \text{ for any } v
\end{cases}
\]

where 
- \( C'_i \) is a cube obtained from \( C' \) by setting to 1 all 
  - \( a_i = 0^j \), for which \( a_i \# b_i = y \),
and
- \( C''_i \) is a cube obtained from \( C'' \) by setting to 1 all 
  - \( a_i = 0^j \), for which \( a_i \# b_i = y \).

**Definition 4.** The \(S\)-operation between two cubes \( C' \) and \( C'' \) is defined as

\[
C' \circ S C'' = \begin{cases} 
\Phi & \text{if } a_i \# b_i = z \text{ for all } i \\
S_j & \text{if } a_i \# b_i = y \text{ for any } i \\
C' & \text{otherwise}
\end{cases}
\]

where \( \Phi \) is the empty set, \( S_j = S_{j-1} \circ_j C' \), \( S_1 = C' \circ_1 C' \), and \( j = 2, 3, \cdots, k \) (\( k \) is the number of steps in which any symbol(s) 0 is generated) while \( C' = c_1 c_2 \cdots c_i \cdots c_n \) and \( c_i = a_i \# b_i \).

According to **Definition 4** it follows that if we substitute the \#-operation in (1) by the \(S\)-operation, the set \( S_m \), consisting of disjoint cubes only, will be generated. The probability of the event, corresponding to a cube \( C \) in \( S_m \), can be calculated as

\[
P(C) = P \cdot Q \prod_{j=1}^{k} (1 - P_j)
\]

where
- \( P = \Pi p_i \) for all \( i \) satisfying \( c_i = 1 \)
- \( Q = \Pi q_i \) for all \( i \) satisfying \( c_i = \text{unique 0} \)
- \( P_j = \Pi p_i \) for all \( i \) satisfying \( c_i = 0^j \)
- \( q_i = 1 - p_i \)

and \( p_i \) is the probability that the \( i^{th} \) element is up.

Now we can propose the algorithm PROB for terminal reliability computation:

**ALGORITHM PROB:**

1. Find path identifiers for all simple paths between node \( s \) and node \( t \)
2. Sort them by increasing weight (where weight = number of 1's in the path identifier string)
3. Set \( P_s = P(IP_1) \)
4. (loop) For \( m = 2, 3, \cdots, k \) determine
   \[
   S_m = (\cdots ((IP_m S IP_1) S IP_2) \cdots) S IP_{m-1}
   \]
   \[
P_m = P_m + P(S_m)
   \]
5. END

In the algorithm, \( P(S_m) \) is the sum of the probabilities of all subcubes in \( S_m \).

As an example, the PROB algorithm is applied to the simple bridge network given in Figure 1. (Figure 2 in Lin\(^{(11)}\) and Figure 1 in Fratta\(^{(10)}\).)

![Figure 1. A Bridge Network](image)

Table 2 presents the set of all simple paths and their path identifiers (for simplicity we consider the case with perfect nodes and imperfect links).

<table>
<thead>
<tr>
<th>( m )</th>
<th>( \pi_m )</th>
<th>( IP_m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( x_1 x_2 )</td>
<td>( 11xx )</td>
</tr>
<tr>
<td>2</td>
<td>( x_3 x_4 )</td>
<td>( xx11x )</td>
</tr>
<tr>
<td>3</td>
<td>( x_1 x_3 x_4 )</td>
<td>( 11x11 )</td>
</tr>
<tr>
<td>4</td>
<td>( x_3 x_5 x_4 )</td>
<td>( x11x1 )</td>
</tr>
</tbody>
</table>

**steps 1,2.** Table 2 presents the set of all simple paths and their path identifiers (for simplicity we consider the case with perfect nodes and imperfect links).

**step 3.** \( P_s = [p_1 p_2] \)

**step 4.** \( m = 2: \)

\[
x11x
\]
\[
11xxx
\]
\[
S_2 = 0011x
\]
\[
j = 1
\]
\[
P_m = P_m + [p_3 p_4 (1 - p_1 p_2)]
\]

48
$m = 3$:

$x1x11$

$11xxx$

$10x11$

$xx11x$

$x111x$

$10011$

$S_3 =$ unique 0, $j = 1$

$P_{m} = P_{m} + [p_1 p_4 p_5 q_3 q_3]$

$m = 4$:

$x11x1$

$11xxx$

$011x1$

$xx11x$

$01101$

$1xx11$

$S_4 =$ unique 0, $j = 2$

$P_{m} = P_{m} + [p_2 p_3 p_5 q_1 q_4]$

step 5.

END

The reliability evaluation requires the computation of 38 intermediate terms, while there are 72 terms in Abraham$^{20}$. Furthermore, the algorithm PROB requires 34 comparisons for the determination of $S_3$ while the algorithms reported in Abraham$^{20}$ and Aggarwal$^{19}$ require 54 and 173 comparisons respectively. For finding $S_{24}$, these numbers are 26, 45 and 159 respectively.

The algorithm was coded in FORTRAN IV and run on a DEC-10 computer system. The execution of algorithm PROB for a 9 node, 13 link example (given in Figure 7 in Lin$^{11}$) took 0.76 seconds. The same example required 71 seconds execution time on a faster machine (namely, a CDC 6500) using the algorithm reported in Lin$^{11}$. For the examples given in Figures 10 and 11 in Fratta$^{23}$ the computing times with PROB were 0.88 and 1.41 seconds. The execution of the algorithm in Fratta$^{23}$, which was coded in ALGOL W on an IBM 360/67 computer, took 4 and 15 seconds respectively.

III. Concluding Remarks

The paper presents a new algorithm for network terminal reliability. The proposed algorithm is based on the implementation of the introduced $S$-operation on the set of the path identifiers only. By application of the $S$-operation, only the disjoint subcubes are generated. The probability of the corresponding event can be calculated in a straightforward manner. The algorithm is efficient since it does not generate a large number of terms, and there is no need for the generation and storage of new terms other than the path identifiers. Since determination of the set $S_m$ is easy, the algorithm can efficiently handle medium size networks. It can be applied to both oriented and non-oriented networks and can be easily modified to produce symbolic expressions for terminal reliability.

The comparison with existing algorithms on the basis of the number of terms, the number of operations and program execution time confirms the superiority of the proposed algorithm over path enumeration algorithms as well as algorithms employing other techniques.

REFERENCES


