A BDD-Based Algorithm for Reliability Graph Analysis*

Xinyu Zang, Hairong Sun and Kishor S. Trivedi
{xzang, hairong, kst@ee.duke.edu}
Center for Advanced Computing and Communications
Department of Electrical and Computer Engineering
Duke University
Durham, NC 27708

Abstract

In this paper, a new algorithm based on Binary Decision Diagram (BDD) for reliability graph analysis is proposed. The reliability graphs are encoded into a BDD that can be easily evaluated. The minpaths and mincuts of a reliability graph can be obtained by BDD. Sensitivity analysis and ordering of variables are discussed in this paper as well. Due to the feature of BDD, the sum of disjoint products (SDP) can be represented by BDD implicitly, which avoids huge storage and high computation complexity for large reliability graphs. Some examples and experiments show the efficiency of this algorithm.

Index Terms: Reliability Graph, Reliability Evaluation, Minpaths, Mincuts, Sensitivity Analysis, Binary Decision Diagram (BDD)

*This research was supported in part by the National Science Foundation under Grant No. EEC9418765, and by the Department of Defense as an enhancement project to the Center for Advanced Computing and Communications in Duke University.

1 Introduction

Reliability evaluation of a system is an important issue in system design, manufacture and maintenance. Many models have already been developed to address this issue. Reliability graph [17] is the one of the most commonly used models.

The *reliability graph* G model consists of a nonempty set $N(G)$ of *nodes*, a set $E(G)$ of directed *edges*, an incidence relation which associates with each edge of G a pair of nodes of G, called its *ends*, where the edges represent components that can fail or structural relationships between the components. The graph contains one node, the *source* (S), with no incoming edges and one node, the *sink* (T) (also called *destination* or *termination*) with no outgoing edges. A system represented by a reliability graph fails when there is no path from the source to the sink. The edges can be assigned failure probabilities, failure rates or unavailability values or functions. A *path* is defined as a set of edges (components) so that if these edges are all up, the system is up. A path is minimal if it has no proper subpaths. The *minpaths* is the set of all minimal paths. A *cut* is defined as a set of edges (components) so that if these edges are all down, the system is down, A cut is minimal if it has no proper subcuts. The *mincuts* is the set of all minimal cuts. The left of Fig. 1 shows an example of reliability model. The system it represents is operational if there is a path from S to T.

Conventionally, two classes of methods are often used for reliability graph analysis[17, 3]. One is factoring algorithm [12, 18, 24]. The idea is to choose an edge (component) and break the model down into two cases: the first assumes the component has failed, the second assumes it has not failed. For each case, a new reliability graph is built by taking into account the behavior of the chosen edge. For example. for the reliability graph at the left of Fig. 1, we choose the edge e_3, the two resulting reliability graphs are shown in the right of Fig. 1. Both of them are series-parallel, so the formulas for series-parallel reliability block

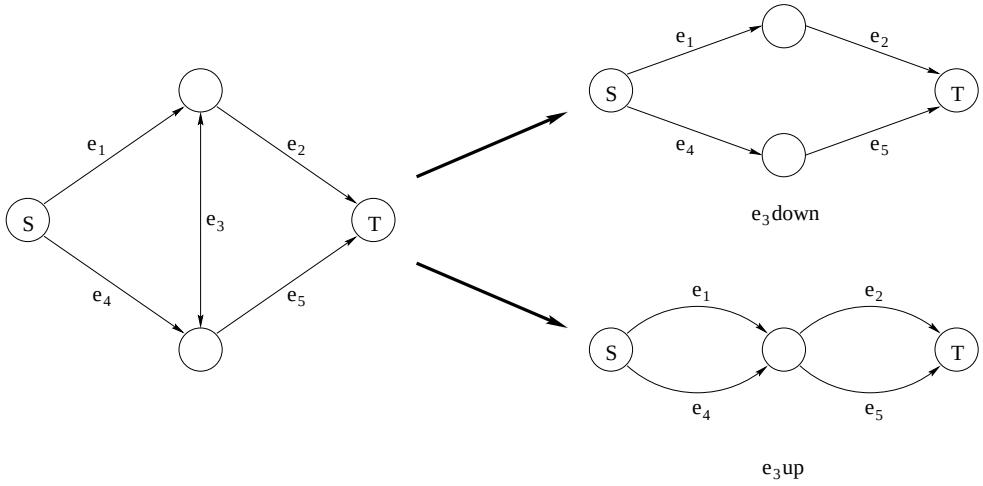


Figure 1: A reliability graph and its factors

diagrams can be applied for their analysis.

The alternative class of methods is to directly obtain minpaths or mincuts. The *inclusion-exclusion*[14, 7, 8, 19] or *sums of disjoint products*(SDP) [15, 23, 10, 6, 22] methods have to be applied to obtain a correct reliability expressions. [9] shows that SDP method has distinct advantages over the inclusion-exclusion method. The minpaths and mincuts of the reliability graph in Fig. 1 are:

- minpaths: $\{e_1, e_2\}$, $\{e_4, e_5\}$, $\{e_1, e_3, e_5\}$, $\{e_4, e_3, e_2\}$.
- mincuts: $\{e_1, e_4\}$, $\{e_2, e_5\}$, $\{e_1, e_3, e_5\}$, $\{e_4, e_3, e_2\}$.

However, there are some disadvantages in the above methods. In the factoring algorithm, the number of factored reliability graphs will increase exponentially with the number of edges increases. In the minpaths or mincuts methods, as the number of edges becomes large, the number of minimal paths or cuts will be large and SDPs or inclusion-exclusion expression for minpaths or mincuts will be increased too large to be stored and be processed.

Binary Decision Diagram (BDD) was, at first, used in VLSI design and verification as an efficient method to manipulate the Boolean expression [2, 1]. [2] and other researches showed that, in most cases, BDDs use less memory to represent large Boolean expressions than representing them explicitly. Because BDDs are based on Shannon's decomposition, reliability evaluation is very easily obtained from BDD format. Some researchers have already used BDD to do reliability analysis for fault tree [16, 4, 20, 21, 5].

In this paper, a new algorithm based on BDD is proposed for reliability graph analysis. A BDD will be generated for a reliability graph, and if necessary, the BDDs that represent minpaths and mincuts can be generated as well. The BDD of the system can represent the SDPs implicitly, avoiding the huge storage for large number of SDPs. Order of variables in BDD will be discussed in this paper as well, because the size of BDD heavily depends on this order. Several ordering heuristics will be provided and compared. Sensitivity analysis will be made for reliability graph.

The paper is organized as follows. Section 2 presents some preliminary concepts of BDD. Section 3 gives the description of the algorithm. Some examples are provided in Section 4 to show the efficiency of the algorithm, and presents the advantage of BDD algorithm by comparison with SDP approach. The last section gives the conclusion and future work.

2 Binary Decision Diagrams (BDD)

In [2], Bryant gave the basic definitions for BDD (also known as function graph), and a subset of general BDD, reduced ordered binary decision diagram (ROBDD) was introduced as an efficient means to manipulate the Boolean expressions. [16, 4, 20] applied BDD to reliability evaluation for fault trees. Here we will review some basic concepts of BDD.

2.1 Shannon's decomposition and *ite* format

Theorem 1 *Shannon's Decomposition: let f be a Boolean expression on X, and x be a variable of X, then,*

$$f = x \cdot f_{x=1} + \bar{x} \cdot f_{x=0} \tag{1}$$

where f evaluated in $x = v$ is denoted by $f_{x=v}$.

Shannon's decomposition is the base of BDD. In order to express Shannon's decomposition concisely, the If-Then-Else (*ite*) format is defined as:

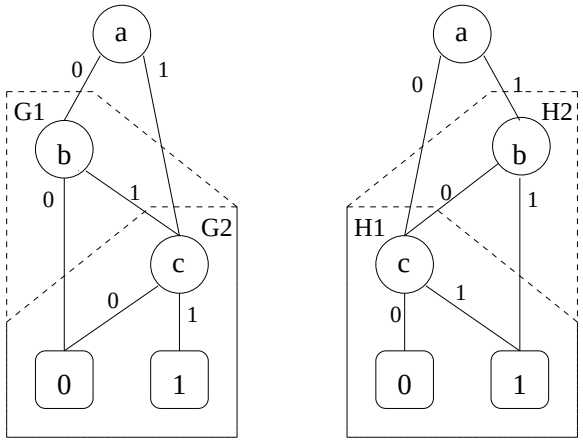
$$f = ite(x, F_1, F_2) = x \cdot F_1 + \bar{x} \cdot F_2 \tag{2}$$

where $F_1 = f_{x=1}$ and $F_2 = f_{x=0}$.

2.2 BDD

A BDD is a directed acyclic graph (DAG) that bases on Shannon's decomposition. The graph has two sink nodes labeled 0 and 1 representing the two corresponding constant expressions. Each non-sink node is labeled with a Boolean variable x and has two out-edges. These two edges are called 0-edge (or *else*-edge) and 1-edge (or *then*-edge). The node linked by 1-edge represents the Boolean expression when $x = 1$, i.e., $f_{x=1}$ in Eqn. (1), while the node linked by 0-edge represents the Boolean expression when $x = 0$, i.e., $f_{x=0}$ in Eqn. (1). Actually, each non-sink node in BDD encodes an *ite* format. Obviously, one of the feature of BDD is its instinctive disjoint.

An ordered binary decision diagram (OBDD) is a BDD with the constraint that the variables are ordered and every source to sink path in the OBDD visits the variables in ascending order. A reduced ordered binary decision diagram (ROBDD) is an OBDD where each node represents a distinct Boolean expression.



(a) $g = a \cdot c + b \cdot c$ (b) $h = a \cdot b + c$

Figure 2: BDD representation of Boolean expressions

In practice, ROBDDs are widely used. Actually, to generate a ROBDD, the ordering of the variables has to be made first and this order of variables will not be changed during the generation[1]. In this paper, we denote $x_i < x_j$ as variable x_j is behind variable x_i in the order of variables.

Fig. 2 shows the ROBDD for several Boolean expressions.

2.3 Manipulation of BDDs

BDD can represent the Boolean expressions graphically. The manipulations of BDD such as logical operations are very easy. For instance, considering a logic operation AND on two Boolean expressions g and h, we generate two BDDs for g and h respectively on the same variable order at first. Then we assume that variable x exists in both expressions. Using

[1] We do not consider the dynamic reordering of BDD in this paper

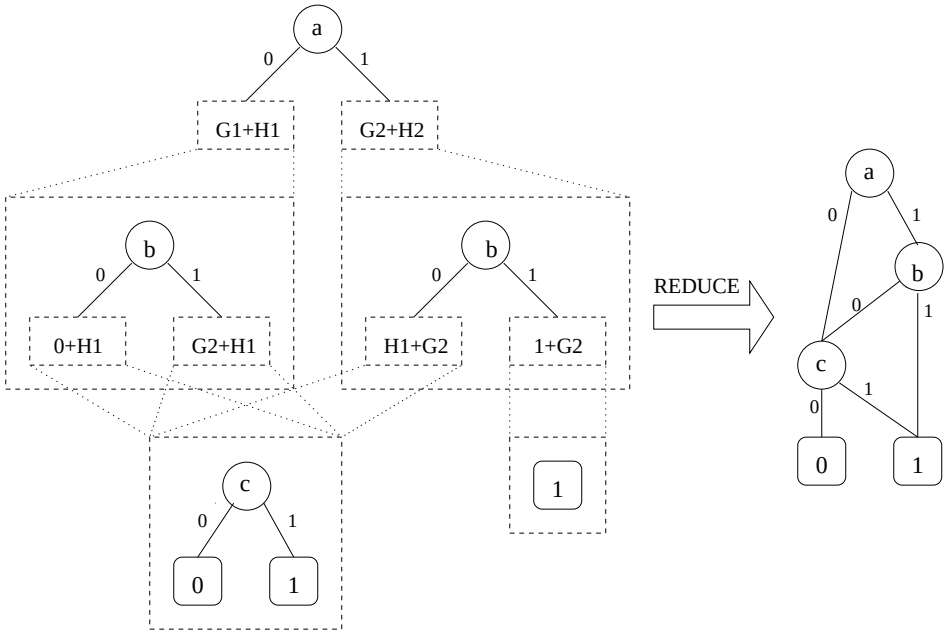


Figure 3: OR operation of two BDDs

Eqn. (2), the *ite* formats of the expressions are

$$g = ite(x, g_{x=1}, g_{x=0}) = ite(x, G_1, G_2)$$

$$h = ite(x, h_{x=1}, h_{x=0}) = ite(x, H_1, H_2)$$

The $g \cdot h$ represented by ite format will be:

$$\begin{aligned} ite(x, G_1, G_2) \cdot ite(x, H_1, H_2) &= g \cdot h \\ &= ite(x, (g \cdot h)_{x=1}, (g \cdot h)_{x=0}) \\ &= ite(x, (g_{x=1} \cdot h_{x=1}), (g_{x=0} \cdot h_{x=0})) \\ &= ite(x, (G_1 \cdot H_1), (G_2 \cdot H_2)) \end{aligned} \tag{3}$$

The recursive method can be used for $G_1 \cdot H_1$ and $G_2 \cdot H_2$ till one of them becomes a constant expression, 0 or 1.

Next we assume h does not have variable x, but have y, and $x < y$ in order of variables. The *ite* formats of the expressions are

$$g = ite(x, g_{x=1}, g_{x=0}) = ite(x, G_1, G_2)$$

$$h = ite(y, h_{y=1}, h_{y=0}) = ite(y, H_1, H_2)$$

The $g \cdot h$ represented by ite format will be:

$$\begin{aligned} ite(x, G_1, G_2) \cdot ite(y, H_1, H_2) &= g \cdot h \\ &= ite(x, (g \cdot h)_{x=1}, (g \cdot h)_{x=0}) \\ &= ite(x, (g_{x=1} \cdot h), (g_{x=0} \cdot h)) \\ &= ite(x, (G_1 \cdot h), (G_2 \cdot h)) \end{aligned} \tag{4}$$

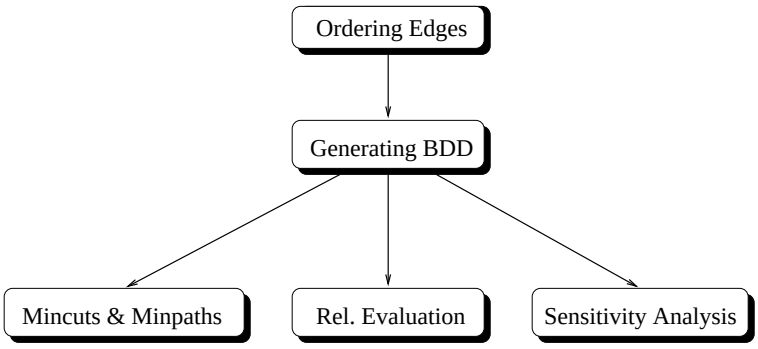


Figure 4: BDD algorithm for reliability graph analysis

In the above example, we used AND operation, but almost every logical operation can be used and the only difference is to use the different truth tables when one of operands becomes constant expression. Actually, in practice, the BDD is generated by using logical operation on variables rather than using Shannon's decomposition directly.

Fig. 3 shows the OR operation of two Boolean expressions in Fig. 2.

3 BDD algorithm for reliability graph analysis

Fig. 4 shows the main procedures of BDD algorithm for reliability graph analysis. First, the edges are ordered by using a heuristic. Then a BDD is generated to encode the reliability graph. From this BDD, many measures can be obtained, e.g. mincuts, minpaths, unreliability, sensitivity analysis.

3.1 Generation of BDD

Fig. 5 gives the algorithm for generating the BDD for a reliability graph.

The function *bdd_gen()* starts from the source node (S), and *this_path* is set to be 0 that

```
bdd_gen(start_node) {
    T_bdd = 0
    set start_node in this_path
    for (edge_i in the set of edges starting from start_node) {
        next_node = the other end of edge_i
        if (next_node == sink_node)
            subpath_bdd = edge_i_bdd
        else if (next_node is already in this_path)
            continue;
        else
            subpath_bdd = bdd_gen(next_node) AND edge_i_bdd
        T_bdd = T_bdd OR subpath_bdd
    }
    clear start_node in this_path
    return T_bdd
}
```

Figure 5: Algorithm for generating BDD from reliability graph

means no node in path. The BDD is returned when *bdd_gen()* finishes. Let Boolean variable E_i represent the edge (component) e_i, and let $E_i = 1$ represent the edge is up, then the $E_i = 0$ means the edge is down. This algorithm actually generates the BDD representation of Boolean expression that represents the system stucture. For the example in Fig. 1, the BDD obtained from *bdd_gen*(S) represents the Boolean expression:

$$E_1 \cdot (E_2 + E_3 \cdot E_5) + E_4 \cdot (E_5 + E_3 \cdot E_2)$$

Fig. 6 shows the BDD generated from reliability graph in Fig. 1.

3.2 Minpaths and mincuts

In [16], Rauzy introduced a BDD operator, '\'(*without*), and a minimal solution method to obtain the minimal BDD that represents the mincuts for fault tree. Here we extend this

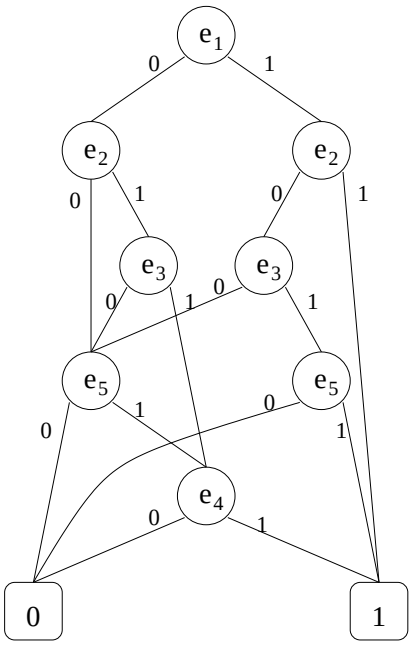


Figure 6: BDD of reliability graph in Fig. 1

method to obtain both mincuts and minpaths from the original BDD.

Fig. 7 shows the algorithm of the extended '\'(*without*) operator. Besides two operands, a flag is needed for indicating whether *path-to-0* or *path-to-1* should be removed. Fig. 8 shows the algorithm of computing the minimal BDD for minpaths and mincuts. The detail proof for this method was given in [16]. The function of the *computed-table* and *unique-table* are explained in [1].

Fig. 9 shows the minimal BDD for minpaths and micuts of the reliability graph in Fig. 1.

- Mincuts: The mincuts is the set of paths in minimal BDD that starts from the root node and proceeds down through the BDD to a sink node 0 and only the variables for which the path leaves their node on a 0-edge on the way to the sink node 0 are contained in path. From Fig. 9(a), we can obtain the mincuts as: $\{e_1, e_4\}$, $\{e_2, e_5\}$, $\{e_1, e_3, e_5\}$, $\{e_4, e_3, e_2\}$.

- Minpaths: The minpathts is the set of paths in minimal BDD that starts from the root node and proceeds down through the BDD to a sink node 1 and only the variables for which the path leaves their node on a 1-edge on the way to the sink node 1 are contained in path. From Fig. 9(b), we can obtain the minpaths as: $\{e_1, e_2\}$, $\{e_4, e_5\}$, $\{e_1, e_3, e_5\}$, $\{e_4, e_3, e_2\}$.

3.3 Evaluation

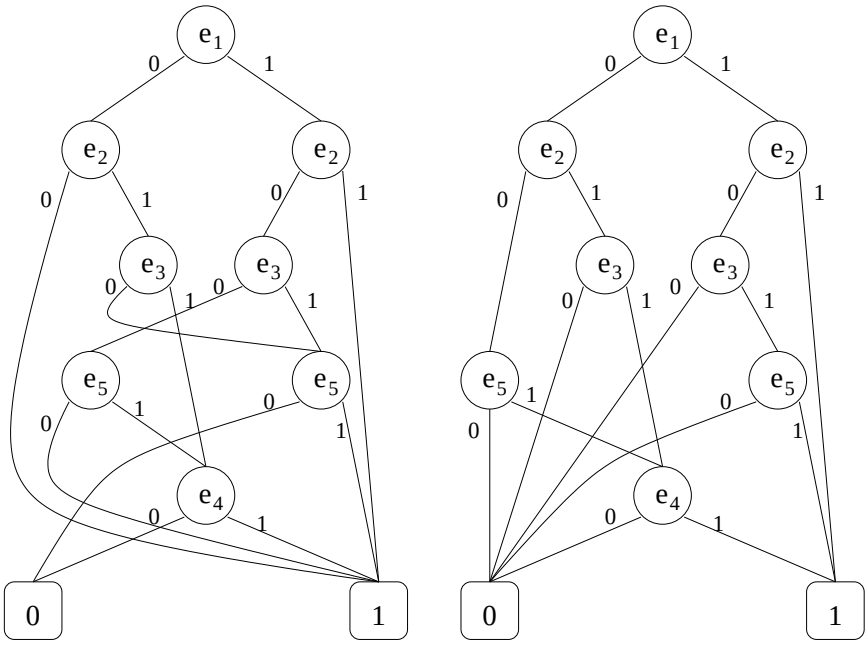
Because BDD is based on Shannon's decomposition, if the probability is assigned to each variable, the performance indices of the system represented by the BDD can be easily obtained. In the introduction, we mentioned that in reliability graph each edge is assigned a failure probability, denoted as $p_{E_i} = P(\bar{E}_i)$. For a BDD F generated from a reliability

```
without(F, G, flag) {
    if ((F == 0) or (F == 1))
        return F
    else if (G == 1) {
        if (remove path-to-1)
            return 0
        else
            return F
    } else if (G == 0) {
        if (remove path-to-0)
            return 1
        return F
    } if (cpomupted-table has entry {(flag, F, G),R})
        return R
    else { /* F = ite(x, F1, F2), G = ite(y, G1, G2) */
        x = the top variable of F
        y = the top variable of G
        v = top of x and y
        F1 = F(v=1), F2 = F(v=0)
        G1 = G(v=1), G2 = G(v=0)
        if (x > y) {
            if (remove path-to-1)
                return without(F, G2, flag)
            else /*remove path-to-0 */
                return without(F, G1, flag)
        } else { /* x <= y */
            T = without(F1, G1, flag)
            E = without(F2, G2, flag)
            if (T == E)
                return T
            R = find_or_add_unique_table(v, T, E)
            insert_computed_table({(flag, F, G), R})
            return R
        }
    }
}
```

Figure 7: Algorithm for $F \backslash G$ with flag

```
minbdd(F, flag) {
    if ((F == 1) or (F == 0))
        return F
    else if (cpomupted-table has entry {(flag, F),R})
        return R
    else { /* F = ite(x, F1, F2) */
       if (get minbdd for minpaths) {
          K = minbdd(F1, flag)
          T = without(K, F2, path-to-1)
          E = minbdd(F2)
       } else { /* get minbdd for mincuts */
          T = minbdd(F1, flag)
          K = minbdd(F2, flag)
          E = without(K, F1, path-to-0)
       }
   }
   if (T == E)
        return T
   R = find_or_add_unique_table(x, T, E)
   insert_computed_table({(flag, F), R})
   return R
}
```

Figure 8: Algorithm for computing minimal BDD with flag



(a) Minimal BDD for mincuts (b) Minimal BDD for minpaths

Figure 9: Minimal BDD for reliability graph in Fig. 1

graph G, we can obtain the unreliability of the system as:

$$\begin{aligned} P(F) &= P(ite(x, F_1, F_2)) \\ &= P(x \cdot F_1) + P(\bar{x} \cdot F_2) \text{ (Disjoint Property)} \\ &= (1 - p_x)P(F_1) + p_x P(F_2) \text{ (Independent Property)} \\ &= P(F_1) + p_x(P(F_2) - P(F_1)) \end{aligned} \tag{5}$$

Using recursive method, we can calculate $P(F_1)$ and $P(F_2)$ until reaching the sink node, i.e. $F_i = 1$ or $F_i = 0$:

- $F_i = 1$, mean the system or subsystem represented by F_i is always up, so $P(F_i) = 0$, it never fails.
- $F_i = 0$, mean the system or subsystem represented by F_i is always down, so $P(F_i) = 1$, it always fails.

Fig. 10 give the algorithm for evaluation.

3.4 Sensitivity analysis

The purpose of the sensitivity analysis(also known as importance measure) is to obtain the information concerning a component's contribution to the system reliability, which can be immensely used in system design, failure diagnosis and system failure probability minimization. The definition of sensitivity analysis of event in fault tree was given in [13]. In this paper, we use the same definition for sensitivity analysis, and obtain three types of measures for reliability graph.

```
Prob(F) {
    if (F == 0)
        return 1
    else if (F == 1)
        return 0
    else if (computed-table has entry {F, P_F})
        return P_F
    else { /* F = ite(x, F1, F2) */
        P_F1 = Prob(F1), P_F2 = Prob(F2)
        P_F = P_F1 + P(x) * (P_F2 - P_F1)
    }
    insert_computed_table({F, P_F})
    return P_F
}
```

Figure 10: Algorithm for evaluation

3.4.1 Birnbaum's importance measure

The Birnbaum's measure of importance provides the probability that the system is in a critical state with respect to component k and that the failure of component k will then cause the system to fail. It is defined as the partial derivative of system unreliability with respect to the failure probability of component k, i.e.

$$I_k^B(t) \stackrel{\mathrm{d}}{=} \frac{\partial F_{sys}(t)}{\partial F_k(t)} = P[\bar{r}(1_k, \bar{\mathbf{x}}) = 1] - P[\bar{r}(0_k, \bar{\mathbf{x}}) = 1] \tag{6}$$

where $\bar{r}(\bar{\mathbf{x}})$ is the system structure function as a function of state vector $(\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n)$, $\bar{x}_i = 1$ means the component x_i has failed, and

$$(1_k, \bar{\mathbf{x}}) \stackrel{\mathrm{d}}{=} (\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_{k-1}, 1, \bar{x}_{k+1}, \cdots, \bar{x_n})$$

$$(0_k, \bar{\mathbf{x}}) \stackrel{\mathrm{d}}{=} (\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_{k-1}, 0, \bar{x}_{k+1}, \cdots, \bar{x_n})$$

$\bar{r}(\bar{\mathbf{x}}) = 1$ represents the system has failed. $F_{sys}(t)$ and $F_k(t)$ are the failure function of the system and component k.

```
CrProb(F, xk, flag) { /* F = ite(x, F1, F2) */
    if (F == 1)
        return 0
    else if (F == 0) {
        if (flag == 0)
            return 0
        else
            return 1
    }
    else if ((x > xk) and (flag == 0))
        return 0
    else if (cpomupted-table has entry {(flag, xk, F),R})
        return R
    else {
        if (x = xk) {
            P_F1 = CrProb(F1, xk, 1), P_F2 = CrProb(F2, xk, 1)
            P_F = P_F2 - P_F1
        } else {
            P_F1 = CrProb(F1, xk, flag), P_F2 = CrProb(F2, xk, flag)
            P_F = P_F1 + P(x) * (P_F2 - P_F1)
        }
    }
    insert_computed_table({(flag, xk, F), P_F})
    return P_F
}
```

Figure 11: Algorithm for Birnbaum's importance measure

There are two ways to calculate the Birnbaum's importance measure. One is directly using the definition formula Eqn. (6): evaluating the BDD by setting failure probability of component k with 1 and 0, then the difference of these two evaluations is the Birnbaum's importance measure. This method need to traverse the BDD twice. The alternative method is using the algorithm shown in Fig. 11, and only need to traverse the BDD once.

3.4.2 Criticality importance measure

The criticality importance measure gives the probability of a component k being responsible for system failure before time t. It is defined as

$$I_k^{CR}(t) \stackrel{\mathrm{d}}{=} \frac{\partial F_{sys}(t)}{\partial F_k(t)} \cdot \frac{F_k(t)}{F_{sys}(t)} = I_k^B(t) \cdot \frac{F_k(t)}{F_{sys}(t)} \tag{7}$$

In order to calculate the criticality importance measure, it only need to follow the Eqn. (7): use the algorithm shown in Fig. 11 to obtain the $I_k^B(t)$ first, then multiply $\frac{F_k(t)}{F_{sys}(t)}$.

3.4.3 Structural importance measure

The structural importance measure allow us to consider the relative importance of various components when only the structure of the system is known. It is defined as:

$$I_k^{ST} \stackrel{\mathrm{d}}{=} \frac{1}{2^{n-1}} \sum_{\bar{\mathbf{x}}} [\bar{r}(1_k, \bar{\mathbf{x}}) - \bar{r}(0_k, \bar{\mathbf{x}})] \tag{8}$$

where $\bar{r}(\bar{\mathbf{x}})$ is the system structure function.

Eqn. (8) gives the definition of structural importance but it is hard to use this formula to compute the structural importance measure because there are 2^{n-1} state vectors have to be evaluated. An alternative method is to assign a failure probability of 0.5 for all components x_i ($i = 1, 2, \ldots, n$ and $i \neq k$), and then use the algorithm shown in Fig. 11. The proof of this method was given in [13].

3.4.4 An example

The Table 1 shows the importance measures for the reliability graph in Fig. 1

Edge(Comp.)	Failure Prob.	Birnbaum's Importance	Criticality Importance	Structural Importance
e_1	0.010	1.02873400e-02	3.10956980e-01	3.75000000e-01
e_2	0.015	1.51905600e-02	6.88751026e-01	3.75000000e-01
e_3	0.020	2.92545000e-04	1.76856038e-02	1.25000000e-01
e_4	0.010	1.02873400e-02	3.10956980e-01	3.75000000e-01
e_5	0.015	1.51905600e-02	6.88751026e-01	3.75000000e-01

Table 1: Sensitivity analysis of reliability graph in Fig. 1

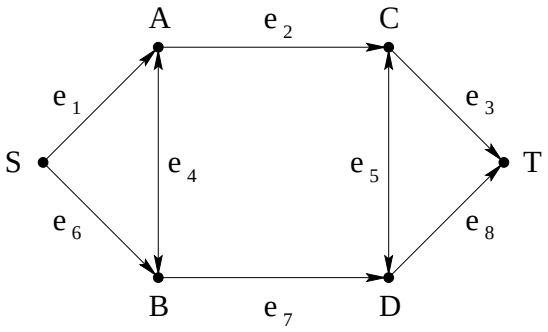


Figure 12: Example of reliability graph

3.5 Ordering Strategies

The order of the variables is very important for BDD generation. The size of BDD (the number of nodes in BDD) heavily depends on the order. But the problem of computing an ordering that minimizes the size of BDD is itself a co NP-complete problem. The previous study showed that a set of heuristics may be used to select an adequate ordering [2, 11], however all these heuristics are proposed for digital circuits that consist of logic gates and there is no similarity between the reliability graph and logic circuits. New ordering strategies are needed for reliability graph analysis. In this section, we present several strategies for ordering the edges. In order to explain the ordering algorithms, we use the reliability graph in Fig. 12 as an example.

3.5.1 Ordering strategy 1

Here we use a stack to implement the order list, while the other three algorithms use a queue to implement the order list.

1. First we search for a path from the source node S to the sinking node T, e.g., we get the result as $\{e_6,\ e_7,\ e_5,\ e_3\}$. Only after e_3 is selected can we be sure that this is a path from the source to the sinking node.

2. Now, we jump back to node C and push e_3, which can reach the sinking node, into the order stack. From node C, we fail to find an alternative path replacing edge e_3.

3. Next we jump back to node D and push e_5 into the stack since via e_5 we can reach node C and then the sinking node. From node D, we search for an alternative path replacing $\{e_5,\ e_3\}$. The searching result is edge e_8. Then we push e_8 into the stack.

4. Next we jump back to node B and push e_7 into the stack since via e_7 we can reach D and then the sinking node. From node B, we search for an alternative path. We find via $\{e_4,\ e_2\}$ we can reach C from which we reach the sinking node. Then we push e_2 and e_4 into the stack subsequently.

5. Finally, we jump back to the source node and try to find an alternate path. Via e_1 we can reach node A then C, which path has been found at the last steps.

The key of this algorithm is: searching a path, pushing the edge which can reach the sinking node into the stack, and then back jumping along the selected path to search for an alternative path. Repeat the procedure till all the edges are included in the stack.

The resulting order of the example in Fig. fg:rgex2 is $e_1 < e_6 < e_4 < e_2 < e_7 < e_8 < e_5 < e_3$. The detailed implementation is illustrated in Fig. 13. In order to make the paper

```
ordering(start_node) {
    set start_node in this_path
    result = 0 /* set if having a path from this node to sink */
    for (edge_i in the set of edges starting from start_node) {
        if (edge_i is already in order_stack)
            continue;
        next_node = the other end of edge_i
        if (next_node == sink_node) {
            push(edge_i, order_stack)
            result = 1;
        }
        else if (next_node is already in this_path)
            continue;
        else {
            if (ordering(next_node) {
                push(edge_i, order_stack)
                result = 1
            }
        }
    }
    clear start_node in this_path
    return result
}
```

Figure 13: Algorithm for ordering edges

concise, we only present the implementation of ordering strategy 1 in this paper.

3.5.2 Ordering strategy 2

1. Starting from the source node S, traversing the graph to search the paths. E.g., we find via $\{e_1, e_2, e_3\}$, the sinking node T is reachable from the source node S. We put $\{e_1, e_2, e_3\}$ into queue of the order list. All the subsequent operation will be based on the path $\{e_1, e_2, e_3\}$.

2. Jumping back from the sinking node T along the path $\{e_1, e_2, e_3\}$ to node C and trying to search for an alternative path. We get the result $\{e_5, e_8\}$. Then we put $\{e_5, e_8\}$ into the queue of the order list.

3. Next jumping from node C along the path $\{e_1, e_2, e_3\}$ to node A, and searching for an alternative path. We find that via link $\{e_4, e_7$ we can reach node D, then node T. So we put $\{e_4, e_7\}$ into the queue.

4. Finally jumping from node A along the path $\{e_1, e_2, e_3\}$ to node S, and searching for an alternative path. We get the result as e_6 and put e_6 into the queue.

The key of this algorithm is: searching a path, putting the edge of the path into the queue, and then back jumping along the selected path to search for an alternative path. Repeat the procedure till all the edges are included in the stack. The resulting order of the example is $e_1 < e_2 < e_3 < e_5 < e_8 < e_4 < e_7 < e_6$.

3.5.3 Ordering strategy 3

The key of this ordering strategy is: starting from the source node S, traversing the graph to search the paths, putting all the edges connected to the new node into the queue while

we are searching for a path. This algorithm is different from the above two methods in the aspect of when to generate the queue or stack. The above methods generate the queue or stack after a path has been found while this method generate the queue while searching the path.

1. First we put e_1 and e_6 into the order list, since they are directly connected to the source node S. Then we select e_1 as the path to the sinking node T, and reach node A.

2. Two edges, i.e., e_2 and e_4 are connected to node A. We put them into the queue. Next we select e_2 as the path to the sinking node T, and reach node C.

3. Two edges, i.e., e_3 and e_5 are connected to node C. We put them into the queue.

4. Then we reach the sinking node, and e_8 is put into the order list because it is connected to T.

5. After the sinking node T is reached, we return to the first step. We select e_6 as the path, and reach node B. Then e_7 is put into the order list.

6. Repeat the procedure, till all the edges are included in the order list.

The resulting order of the example is $e_1 < e_6 < e_2 < e_4 < e_3 < e_5 < e_8 < e_7$.

3.5.4 Ordering strategy 4

For each node in the reliability graph, we calculate the minimum number of hops from the node to the sinking node T and assign these values to the corresponding nodes. Then, we put the edges which connected to the node with maximal value into the queue. Next select

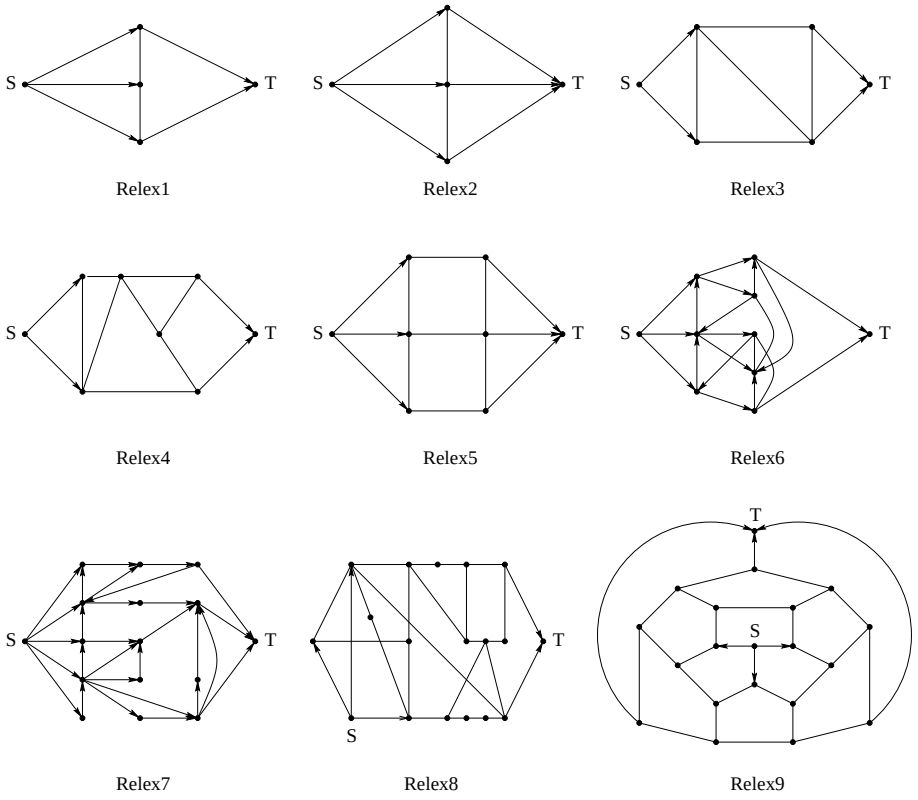


Figure 14: Examples of reliability graph

the node with submaximal value and put the node's attached edges into the queue. Repeat the procedure till the edges connected to node with minimal value are put into the queue.

For the example in Fig. 12, the values for nodes S, A, B, C and D are 3, 2, 2, 1 and 1 respectively. The resulting order of the example is $e_1 < e_6 < e_2 < e_4 < e_7 < e_3 < e_5 < e_8$.

In the next section, we will show the BDD size corresponding to the different orders.

	SDP		BDD (O1)		BDD (O2)		BDD (O3)		BDD (O4)	
Example	# DP	Time(s)	Size	Time(s)	Size	Time(s)	Size	Time(s)	Size	Time(s)
relex1	7	0.03	15	0.04	15	0.04	19	0.04	17	0.04
relex2	11	0.04	27	0.05	19	0.05	27	0.05	27	0.05
relex3	16	0.04	21	0.05	28	0.05	32	0.05	28	0.05
relex4	40	0.05	28	0.05	57	0.05	54	0.05	33	0.05
relex5	78	0.06	64	0.06	65	0.06	85	0.06	67	0.06
relex6	150	0.10	291	0.08	347	0.08	277	0.08	277	0.08
relex7	402	0.31	120	0.06	187	0.07	1178	0.12	444	0.09
relex8	2294	6.46	966	0.16	505	0.19	4865	0.38	743	0.15
relex9	47312	148.45	2083	0.41	14821	2.26	12277	1.25	3360	0.46

Table 2: Experiment results

4 Examples and comparison with SDP based approach

In this section, we use the nine examples given in [10] to compare the four heuristics for ordering and SDP method. Fig. 14 shows these examples[2]. Table 2 shows the experiment results which are generated on Sun Ultra 1 workstation. Observing these results, we can obtain some feature of BDD algorithm:

- BDD algorithm is much faster than SDP algorithm when reliability graph is large.
- In most cases, the size of BDD is not very large if an appropriate order is used.
- The size of BDD heavily depends on the ordering of the variables.

[2]A link without direction arrow is bidirection.

5 Conclusion

We presented a BDD-based algorithm for reliability graph analysis. Several related issues, such as generation of BDD from reliability graphs, order of variables, evaluation of reliability, generation of mincuts and minpaths and sensitivity analysis, were discussed. Due to the feature of BDD, this algorithm is more efficient than the algorithm based on SDP method in both computation and storage, which makes it possible for us to study some practical and large reliability graph.

References

[1] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a bdd package. *Proc. 27th ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[2] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computer*, 35(8):677–691, 1987.

[3] C.J. Colbourn. *The Combinatorics of Network Reliability.* Oxford University Press, New York, 1989.

[4] O. Coudert and J.C. Madre. Metaprime: An interactive fault-tree analyzer. *IEEE Transactions on Reliability*, 43(1):121–127, 1994.

[5] S.A. Doyle and J.B. Dugan. Dependability assessment using binary decision diagrams. *Proc. 25th International Symposium on Fault-Tolerant Computing*, pages 249–258, 1995.

[6] K.D. Heidtmann. Smaller sum of disjoint products by subproduct inversion. *IEEE Transaction on Reliability*, 38:305–311, 1989.

[7] Y.H. Kim, K.E. Case, and P.M.Ghare. A method for computing complex system reliability. *IEEE Transaction on Reliability*, R-21:215–219, 1972.

[8] P.M. Lin, B.J. Leon, and T.C. Huang. A new algorithm for symbolic system reliability analysis. *IEEE Transaction on Reliability*, R-25:2–15, 1976.

[9] M.O. Locks. Recursive disjoint products: a review of three algorithms. *IEEE Transaction on Reliability*, R-31:33–35, 1982.

[10] T. Luo and K.S. Trivedi. An improved algorithm for coherent-system reliability. *IEEE Transaction on Reliability*, 47(1):73–78, 1998.

[11] M.Bouissou. An ordering heuristic for building binary decision diagrams from fault-trees. *Proc. 1996 Annual Reliability and Maintainability Symposium*, pages 208–214, 1996.

[12] K.B. Misra. An algorithm for the reliability of redundant networks. *IEEE Transaction on Reliability*, R-19:146–151, 1970.

[13] K.B. Misra. *Reliability Analysis and Prediction: A Methodology Oriented Treatment.* Elsevier, Amsterdam, The Netherlands, 1992.

[14] K.B. Misra and T.S.M. Rao. Reliability analysis of redundant networks using flow graphs. *IEEE Transaction on Reliability*, R-19:19–24, 1970.

[15] S. Rai, M. Veeraraghavan, and K.S.Trivedi. A suvey of efficient reliability computation using disjoint products approach. *Networks*, 25:147–163, 1995.

[16] A. Rauzy. New algorithms for fault tree analysis. *Reliability Engineering and System Safety*, 40:203–211, 1993.

[17] R.A. Sahner, K.S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer System.* Kluwer Academic Publishers, 1996.

[18] A. Satyanarayana and M.K. Chang. Network reliability and the factoring theorem. *Networks*, 13:107–120, 1983.

[19] A. Satyanarayana and A. Prabhakar. New toplogogical formula and rapid algorithm for reliability analysis of complex networks. *IEEE Transaction on Reliability*, R-27:82–100, 1978.

[20] R.M. Sinnamon and J.D. Andrews. Improved accuracy in quantitative fault tree analysis. *Quality and Reliability Engineering International*, 13:285–292, 1997.

[21] R.M. Sinnamon and J.D. Andrews. Improved efficiency in qualitative fault tree analysis. *Quality and Reliability Engineering International*, 13:293–298, 1997.

[22] S. Soh and S. Rai. Carel: Computer aided reliability evaluator for distributed computing networks. *IEEE Transaction on Parallel and Distributed Systems*, 2(2):199–213, 1991.

[23] M. Veeraraghavan and K.S. Trivedi. An improved algorithm for symbolic reliability analysis. *IEEE Transaction on Reliability*, 40:347–358, 1991.

[24] R.K. Wood. A factoring algorithm using polygon-to-chain reductions for computing k-terminal network reliability. *Networks*, 15:173–190, 1985.