Finding All Premises of a Propositional Formula Using Binary Decision Diagrams

Satoru Yamaguchi, Madori Ikeda, Keisuke Otaki, Ryo Yoshinaka and Akihiro Yamamoto

Graduate School of Infomatics, Kyoto University, Yoshida-honmachi, Sakyo-ku, Kyoto, JAPAN

Abstract. We propose a method using binary decision diagrams (BDDs) to enumerate all Boolean functions corresponding to all premises of a given formula. The problem of finding premises appears, for example, in inductive logic programming (ILP). Generally, the number of the premises is huge, and previous methods generate solutions under some restrictions or select optimal solutions from given candidates. However, these method might miss some hypotheses to evaluate. Because various fast enumeration algorithms have been developed, we are motivated by trying to develop an efficient algorithm to enumerate all solutions analyzing it about this problem too. We use the data structure called BDDs. BDDs can be regarded as data structures representing values of propositional formulas under their interpretations. Our method treats propositional formulas including its input and outputs in the form of BDDs. Thereby, we can enumerate premises by operating paths of a BDD. The method uses synthesis of BDDs to operate paths and makes a new graph called an intermediate graph to efficiently generate BDDs to be synthesized with the given BDD.

1 Introduction

The problem of finding premises appears in inductive logic programming (ILP), a kind of machine learning which uses features of symbolic logic. In ILP general rules are constructed so that they can explain observed examples. More precisely, with letting B be a formula representing background knowledge, E be a formula representing an observed example, it is required to generate a formula, H satisfying $B \wedge H \models E$. This problem is called hypothesis finding. Similarly, Consequence finding is generating E satisfying $B \wedge H \models E$ when B and H are given. Hypothesis finding is also used, for example, to solve a graph completion problem of metabolic pathway [3]. Since the number of solutions of each of these problems increases exponentially, researchers have constructed methods to generate solutions under some restrictions or to select some optimal solutions from given candidates. Various fast enumeration algorithms have been developed in recent years. We are motivated by trying to develop an efficient algorithm and analyzing it on the viewpoint on enumeration.

Because the relation $B \wedge E \models H$ in the definition of hypothesis finding can be transformed so that H is a premise of $\neg B \vee E$, our proposed method enumerates all formulas which is premises of a given formula. Moreover, because a propositional formula can be considered as a logical function whose input is the interpretation and which takes the value of the formula under its interpretation, we give an enumeration of premises by using BDDs, the data structure representing a logical function. The input of our proposed method is the BDD representing a formula and our method enumerates all premises of the input as BDDs. We can efficiently synthesize two BDDs and get a new BDD representing the result of some logical operations of them. Besides, outputs in the form of BDDs are very compact because BDDs can share equivalent subgraphs. BDDs representing premises of a propositional formula tend to resemble each other. Some methods to convert a BDD to a propositional formula have already been proposed [9].

With a BDD, we can know the value of the formula from the leaf node reached by following nodes from the root node according to the input. We can consider that the path from the node to the leaf is corresponding to the input interpretation. We can generate premises of a formula by manipulating the values of the formula under some interpretations. Therefore, we can enumerate premises as BDDs by manipulate some paths of the input BDD. However, we have to solve the problem how to manipulate a node or an edge affects more than one path. Our proposed method manipulates paths by using synthesis of BDDs. We call the BDD to be synthesized with a given BDD a *covering BDD*. We can decide the value of some interpretation of a covering BDD. Because the complexity of synthesis depends on the number of nodes of the BDDs, it is desired to decrease the number of nodes [8]. Our proposed method constructs a new graph named an *intermediate graph* and uses it to efficiently construct the smallest covering BDDs.

The rest of this paper is organized as follows. Section 2 provides our terminology and notation on propositional formulas. Section 3 introduces BDDs as a data structure of propositional formulas and

states that we can generate premises by utilizing synhesis of BDDs. Section 4 explains our proposed method by dividing it into a preprocessing and an enumeration part.

2 Preriminaries

2.1 Propositional Formula

We use $x_1, x_2, \ldots$ as *variables*, $\wedge$ (*conjunction*), $\vee$ (*disjunction*) and $\neg$ (*not*) as *logical connectors*. *propositional formulas* are formulas made by applying logical connectors variables, like $(x_1 \wedge \neg x_2) \vee x_3 \vee (x_2 \wedge x_4)$. We call propositional formulas just formulas for short in this paper.

We use 1 (*true*) and 0 (*false*) as *the Boolean values*. Let F be a propositional formula consisting of $x_1, \ldots, x_n$. *The interpretation* I of F is an assignment of a Boolean value to each variable. The number of interpretation of F is 2^n. A formula F takes a Boolean value corresponding to each interpretation. The value can be determined inductively as follows:

1. $\neg F$ takes 0 if F takes 1. Otherwise it takes 1.
2. $F_1 \wedge F_2$ takes 1 when F_1 takes 1 and F_2 takes 1. Otherwise it takes 0.
3. $F_1 \vee F_2$ takes 0 when F_1 takes 0 and F_0 takes 0. Otherwise it takes 1.

For example, the propositional formula $(x_1 \wedge \neg x_2) \vee x_3 \vee (x_2 \wedge x_4)$ takes 0 under the interpretation $(x_1, x_2, x_3, x_4) = (0, 1, 0, 0)$ and 1 under the interpretation $(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$. When we say a propositional formula F_1, F_2 are different from each other, it means an interpretation I exists such that $I(F_1) \neq I(F_2)$. When a propositional formula F takes a Boolean variable b under its interpretation I, we write it as $I(F) = b$. When $I(F_1) = 1$ for all interpretations I satisfying $I(F_2) = 1$, F_1 is called a *logical consequence* of F_2 and F_2 is called a *premise* of F_1. This relationship is written as $F_2 \models F_1$. Examples of premises of $(x_1 \wedge \neg x_2) \vee x_3 \vee (x_2 \wedge x4)$ are $x_3 \vee (x_2 \wedge x4)$ and $(x_1 \wedge \neg x_2) \vee x_3$. When the number of interpretation satisfying $I(F) = 1$ is m, the number of premises is 2^m.

2.2 Binary Decision Diagrams

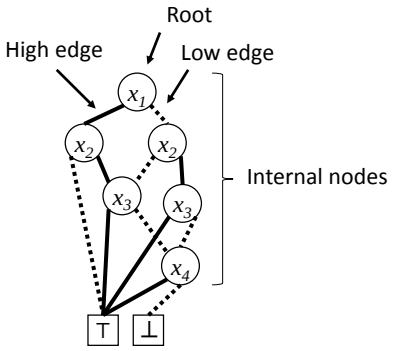

Fig. 1. An example of a BDD.

A BDD G is expressed as a rooted directed acyclic graph $\langle V, E \rangle$ where V is a set of *nodes* and E is a set of *edges*. Edges are directed and there are two kinds of edges, *low edges* and *high edges*. There are two kinds of nodes, *internal nodes* and two *leaves*. A Leaf is either $\top$ or $\bot$. They have no outgoing edges. Each internal node v has two outgoing edges, a low edge and a high edge, and also has a *label*. The label is denoted by label(v)$\in \{1, \ldots, n\}$. A BDD has a unique node called the root which has no incoming edge. For an internal node $v \in V$, low(v) means the node pointed by the low edge of v and high(v) means the node pointed by the high edge of v. For a node v' satisfying $v' = \text{low}(v)$ or $v' = \text{high}(v)$, we call v' a *child* of v and v a *parent* of v'. If $v' = \text{low}(v)$, then we call v' the *low child* of v and If $v' = \text{high}(v)$, then we call v' the *high child* of v. Nodes which can be reached by following children are called *descendants*.

If each internal node u pointing to an internal node v satisfies label(u) $<$ label(v), then the BDD is *ordered*. A BDD is *reduced* if the following *reduction operations* cannot be applied:

1. For each internal node u whose two children are same node v, redirect all the incoming edges of u to v and remove u.
2. For any nodes u and v whose fields are same respectively, redirect all the incoming edges of u to v and remove u.

Fig. 1 shows an ordered and reduced BDD. We call ordered and reduced BDDs just BDDs.

BDDs are used as data structures of logical functions. Let G be a BDD representing a logical function $f(x_1, \ldots, x_n) : \{0,1\}^n \longmapsto \{0,1\}$. Then the number of labels is n, and a label i of each node means the node is corresponding to the variable x_i of the function. We can know the output of the function from the leaf node reached by following internal nodes from the root according to the input: for each internal node v, a node to be followed is high(v) if label(v) takes 1, low(v) otherwise. If the reached leaf is $\top$, the function takes 1 under the input. Otherwise, it takes 0. We can efficiently *synthesize* two BDDs for logical functions, f and g, with logical connectors and obtain new BDDs for functions such as $f \wedge g$ or $f \vee g$ [8].

2.3 Hypothesis Finding and BDDs

The *hypothesis finding* is to generate a formula H satisfying $B \wedge H \models E$ when B and E are given where B represents background knowledge and E represents an observed example. Since the relation between B, H and E can be transformed into $H \models \neg B \vee E$ and B and E are given, we can solve hypothesis finding by generate premises given a logical consequence. Our proposed method enumerates all premises of a given formula.

We can consider a propositional formula F as a logical function whose input is the interpretation I of F and which takes $I(F)$. Therefore, we can use BDDs as a data structure of values of propositional formulas under each interpretation. We indicate a BDD representing F by BDD(F). Then each interpretation is corresponding to a path of the BDD which is followed to know the value. For example, the BDD shown in Fig. 1 represents the propositional formula $(x_1 \wedge \neg x_2) \vee x_3 \vee (x_2 \wedge x_4)$. The interpretation $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$ is corresponding to the path consisting of low edge, high edge, low edge and high edge from the root.

Now, let P and C be propositional formulas satisfying $P \models C$. Then, for a propositional formula F, $C \wedge F = P$ holds if F satsfies the following conditions,

1. for all interpretation I satisfying $I(P) = I(C) = 1$, $I(F) = 1$, and
2. for all interpretation I satisfying $I(P) = 1$ and $I(C) = 0$, $I(F) = 0$.

Therefore, when the logical consequence C is given, for each premise, we can get it by making a formula satisfing these condition. Then we can arbitrarily decide the value of $I(F)$ for the interpretation I satisfing $I(P) = 0$. We can say the same things with BDDs. When BDD(C) is given, for each premise, we can get it by making a BDD(F) so that F satisfies the conditions and by sinthesizing it with BDD(C). We call the BDD synthesized with the given BDD a covering BDD. Let BDD(F) be a covering BDD of BDD(C). The paths of BDD(F) corresponding to the ones of BDD(C) reaching $\bot$ can reach either $\top$ or $\bot$. It is desired to decide which leaf they reach so that the covering BDD is the smallest because the complexity of synthesis of BDDs depends of the number of nodes. We decide the values so that each covering BDD has as many shared nodes as possible.

3 Enumeration of premises by using BDDs

From the definition of premises and the relationship between a logical formula and a BDD equivalent to the function represented by the formula, we can enumerate premises by manipulating some paths from the root to the leaf representing output "1" of the given BDD. However, it is not easy to manipulate nodes and edges of BDDs because such manipulation of a node or an edge affects more than one path. Besides it, since exponential numbers of premises must be enumerated, we must consider the order of enumeration. In order to solve these problems, our proposed method first constructs a new graph named an *intermediate graph* as preprocessing. Second, it enumerates BDDs named covering BDDs and finally obtains premises by computing conjunction of the covering BDDs and the given BDD one by one.

3.1 Intermediate graphs

We define an intermediate graph as the graph made by applying the following operations to a given BDD.

1. Remove all edges connecting to $\bot$.
2. Repeat the following operations until they cannot be applied.
   (a) For an internal node v satisfying label(low(v))-label(v) > 1, make new node u and set high(u) $\leftarrow$ low(v), low(u) $\leftarrow$ low(v), label(u) $\leftarrow$ label(v) $+ 1$, low(v) $\leftarrow u$.
   (b) For an internal node v satisfying label(high(v))-label(v) > 1, make new node u and set high(u) $\leftarrow$ high(v), low(u) $\leftarrow$ high(v), label(u) $\leftarrow$ label(v) $+ 1$, high(v) $\leftarrow u$.
   (c) Apply the second reduction operation.
3. Remove all nodes which have only a child.

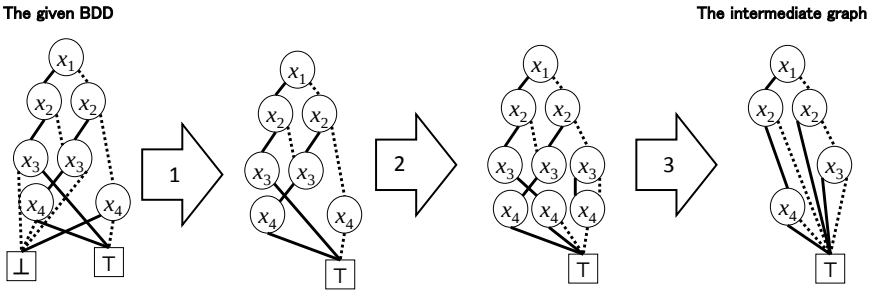

Fig. 2. An example flow of constructing an intermediate graph.

In Fig. 2 we illustrate an example of applications of these operations. The figure show the process to make the right intermediate graph from the left BDD. First, by removing all edges connecting to $\top$ of the BDD, we get the graph on the right of the arrow labeled with 1. The second step is to insert a node between two nodes so that difference of the labels of each node and its child is 1. By this operation, we get the graph on the right of the arrow lebeled with 2. Finally, we get the intermediate graph by removing all nodes which have only a child. Let n be the number of variables of a given BDD and s be the number of nodes of it. In the operations, reduction has the largest time complexity. The reduction of BDD is proportional to the sum of the number of nodes and the number of variables. After the second operation, the number of nodes of the graph is same as that of a quasi-reduced BDD (QDD). BDD to QDD ratio is approximately less than or equal to $1 + O(2^{-\frac{n}{3}}n)$ [4]. Therefore construction of an intermediate graph has a time complexity of $O((1 + 2^{-\frac{n}{3}}n)s + n)$.

3.2 Enumeration using an intermediate graph

A covering BDD is a BDD made by redirecting some edges of an intermediate graph to $\top$. Because each covering BDD is corresponding to a premise one to one, we need to enumerate all covering BDDs, namely, all cases of redirection of edges of an intermediate graph. Our proposed method generates a covering BDD by redirecting edges step by step with referring the previous covering BDD. There are three types of redirection for each internal node:

- Low recursion, where we redirect its high edge to $\bot$,
- High recursion, where we redirect its low edge to $\bot$ and
- Both recursion, where we do not redirect its edges.

If we redirect both edges of an internal node t $\top$, then the node get equivalent to $\bot$. Because the redirection is equivalent to redirection of edges pointing the node to $\bot$, we did not include the redirection in the types. The type transitions in this order with generating covering BDDs. The children pointed by its non-redirected edges are also assigned either type to. In Fig. 3 we illustrate the types of redirection. The left one shows low recursion and its high child points $\bot$. The central one shows high recursion

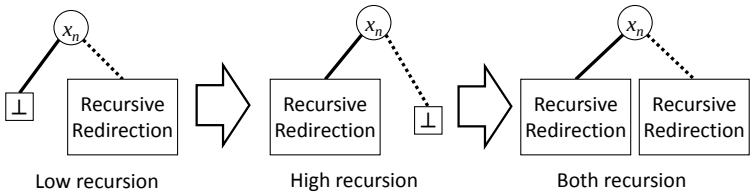

Fig. 3. The order of redirection.

and its low child points $\bot$. The right one shows both recursion. However both children are recursive redirection in the figure, the operations to apply are not symmetry. We are explaining the operations of both redirection later. We decide the type of each node by applying a procedure from the root of the intermediate graph. The operation of the procedure can be divided as follows.

1. Continuous low recursion: assign low recursion to the current node and apply continuous low recursion to the low child of current node.
2. Transition: Transition the type of redirection of the current node and apply continuous low recursion to the child pointed by non-redirected edge.
3. Delegation: Do nothing on this node and apply the procedure to its child.

Let *this node* means the node of the intermediate graph which is applied procedure now and *the previous node* means the node of the previously generated covering graph which can be reached by following same edges as this node. Then we can write the procedure as follows.

1. If this node is $\top$, then do nothing.
2. If the previous node is $\bot$, then apply continuous low recursion to this node.
3. If the previous node is low recursion,
   (a) if its low child is $\top$, then transition the type of this node to high recursion.
   (b) otherwise, delegate to its low child.
4. If the previous node is high recursion,
   (a) if its high child is $\top$, then transition the type of this node to both recursion.
   (b) otherwise, delegate to its high child.
5. If the previous node is both redirection,
   (a) if the low child is $\top$, then apply this procedure to its high child and apply continuous low recursion to its low child.
   (b) otherwise, delegate to its low child.

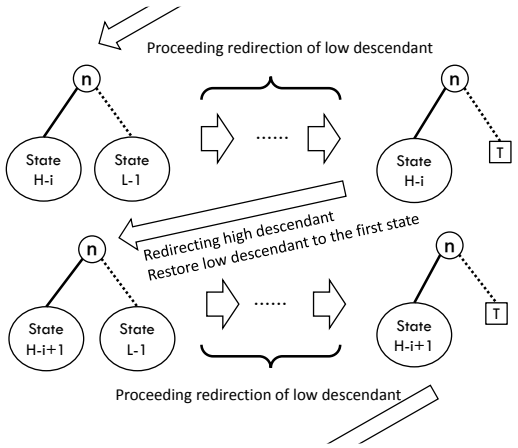

Fig. 4. The detailed order of both recursion

Table 1. The value to substitute for low(v) and high(v). Because low(v_p) cannot point $\top$ when high(v_p) points $\top$ or an internal node, we did not write the case.

low(v_p)	high(v_p)	low(v)	high(v)
An internal node	$\bot$	replace(low(v_i), low(v_p))	$\bot$
$\top$	$\bot$	$\bot$	replace(high(v_i),$\bot$)
$\bot$	An internal node	$\bot$	replace(high(v_i),high(v_p))
$\bot$	$\top$	replace(low(v_i), $\bot$)	replace(high(v_i),$\bot$)
Not $\bot$	An internal node	replace(low(v_i), low(v_p))	high(v_p)
An internal node	$\top$	replace(low(v_i),$\bot$)	replace(high(v_i),high(v_p))

Transition happens when the child to be delegated to and all its descendants are both redirection. Then the child is equivalent to $\top$ and replaced $\top$ by a reduction operation. Therefore we can know when to transition by checking whether a child of the node is $\top$ or not. Since the operation of both recursion is different from those of the other one, We explain about it. In Fig. 4 we illustrate the order of both recursion. In both recursion, we can consider that each subgraph rooted at a high child has a type of redirection. For each subgraph, we redirect edges in the same way as low recursion and we make new subgraph when the low child get to be $\top$. Fig. 5 shows examples of covering graphs generated from an intermediate graph by redirection. Algorithm 1 shows the precise procedure of redirection and Algorithm

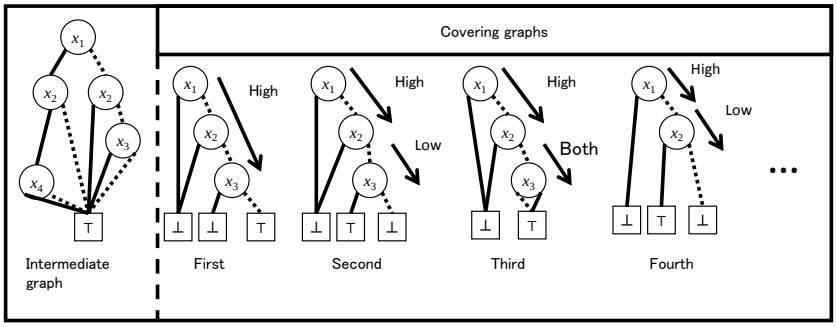

Fig. 5. Examples of covering grpahs: This shows covering graphs generated from the intermediate graph. High, Low and Both on the right of arrows mean the kind of redirection.

2 shows the whole procedure of our proposed method where the function make_intermediate(r) returns the intermediate graph of the BDD whose root is r and the function reduce(r) returns the BDD made by apply reduction operations to the BDD whose root is r. Fig. 6 shows a whole flow of our proposed method. The average delay time of generate a covering BDD is $O(h)$, where h is the number of nodes of the longest path from the root to a leaf. This can be derived by assuming any path of the intermediate graph has the length of h and evaluate the recurrence formula of the number of calling the function replace. The one of the worst case is $O(h^2)$. It is took when the previously generated covering BDD is the following conditions,

1. the root and its descentdant which can be reached by follwing high edge from the root is both redirection.
2. The children pointed by the low edge of them are $\top$.

4 Conclusion

We proposed a method to enumerate premises of a propositinal formula given as a BDD. This method makes an intermediate graph as preprocessing. After that, it enumerates premises by repetition of generating a covering graph and a synthesis of the covering graph and the given BDD.

Algorithm 1 replace(v_i, v_p).

Input: v_i is the pointer to an internal node of the intermediate graph, v_p is th pointer to an internal node of the previous covering BDD.
  if $v_i = \top$ **then**
    return the pointer to $\top$.
  end if
  if label(v_p) $\neq$ label(v_i) **then**
    make a new node and let the pointer to it be v_q.
    label(v_q) := label(v_i)
    low(v_q) := v_p
    high(v_q) := v_p
    $v_p := v_q$
  end if
  make a new node and let the pointer to it be v.
  label(v) := label(v_i)
  if $v_p = \bot$ **then**
    low(v) := replace(low(v_i), $\bot$)).
    high(v) := $\bot$.
  else
    substitute the value for low(v) and high(v) according to Table 1 and Table 2.
  end if
  if low(v) = high(v) **then**
    v := low(v).
  end if
  return v.
Output: the returned node is a node of the covering graph.

Algorithm 2 enumerate(r)

Input: r is the root of a given BDD.
  r_i := make_intermadiate(r)
  $r_p := \bot$
  output and-apply(r, r_p).
  while $r_p \neq \top$ **do**
    r_p := replace(r_i, r_p)
    r_p = reduce(r_p)
    output r_p
  end while
Output: the output nodes are roots of a BDD representing a premise of a given BDD.

Our future work includes to calculate the time complexity more precisely and to evaluate our method empirically. Besides, th We should decrease the time complexity of the worst case $O(h^2)$ by reconsidering the order of enumeration.

References

1. Alvaro del Val: A New Method for Consequence Finding and Compilation in Restricted Languages, AAAI/IAAI, pp.259-264, (1999).
2. Donald E. Knuth: The Art of Computer Programming, Volume 4 Fascicle 1, pp.70-97, (2009).
3. Gabriel Synnaeve, Addrei Doncescu and Katsumi Inoue: kinetic Models for Logic-Based Hypothesis Finding in Metabolic Pathways, The 19th International Conference On Inductive Logic Programming (ILP 2009), (2009).
4. Ingo Wegener: The Size of Reduced OBDDs and Optimal Read-once Breanching Programs for Almost all Boolean Fnctions, Proceeding of the 19th International Workshop (WG '93) Graph-Theoretic Concepts in Science, pp.252-263, (1994).
5. Jinbo Huang, Adnan Darwiche: Using DPLL for efficient OBDD construction, Theory and Applications of Satisfiability Testing Lecture Notes in Computer Science Volume 3542, pp.157-172, (2005)
6. Katsumi Inoue: *Abudakushon No Genri* (The Principle of Abduction), *Journal of Japanese Society for Artificial Intelligence* 7(1), pp.48-59, (1992).
7. Katsumi Inoue: Linear Resolution for Consequence Finding, Artif. Intell. 56(2-3), pp.301-353, (1992)
8. Rondal E. Bryant: Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams, ACM Computing Surveys (CSUR) Volume 24 Issue 3, pp.293-318, (1992).

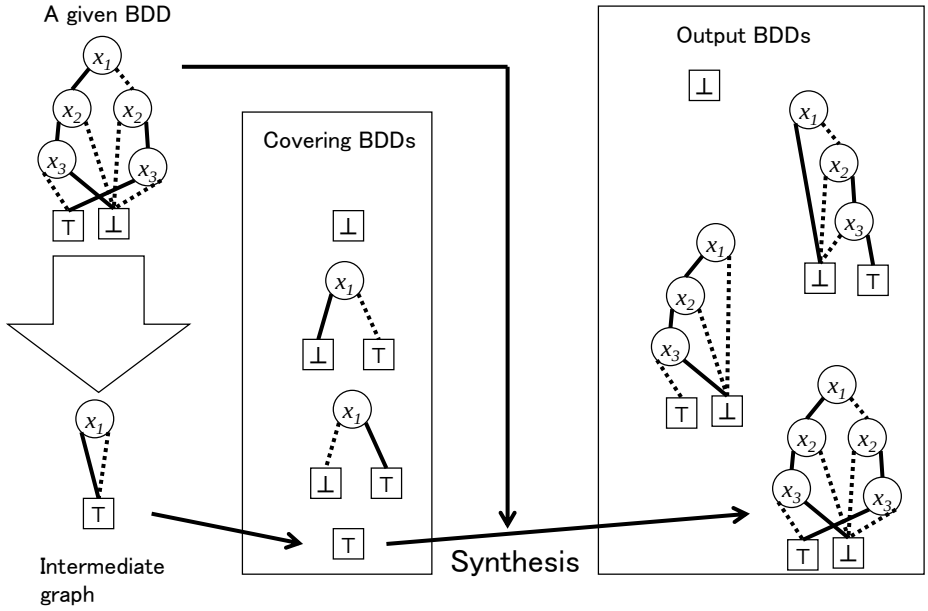

Fig. 6. An example of the whole flow of our proposed method.

9. Shin-ichi Minato: Binary Decision Diagrams and Applications for VLSI CAD, Kluwer Academic, pp.49-60, (1996)
10. Takeaki Uno: A Fast Algorithm for Enumerating Bipartite Perfect Matchings, Lecture Note in Computer Science 2223 (International Symposium on Algorithm and Computation), pp.367-379, (2001).