Cryptogr. Commun.
https://doi.org/10.1007/s12095-018-0304-7

Factorization using binary decision diagrams

Håvard Raddum[1] · **Srimathi Varadharajan**[1]

Received: 23 November 2017 / Accepted: 10 April 2018

Abstract We address the factorization problem in this paper: Given an integer $N = pq$, find two factors p and q of N such that p and q are of same bit-size. When we say integer multiplication of N, we mean expressing N as a product of two factors p and q such that p and q are of *same bit-size*. We work on this problem in the light of Binary Decision Diagrams (BDD). A Binary Decision Diagram is an acyclic graph which can be used to represent Boolean functions. We represent integer multiplication of N as product of factors p and q using a BDD. Using various operations on the BDD we present an algorithm for factoring N. All calculations are done over $GF(2)$. We show that the number of nodes in the constructed BDD is $\mathcal{O}(n^3)$ where n is the number of bits in p or q. We do factoring experiments for the case when p and q are primes as in the case of RSA modulus N, and report on the observed complexity. The multiplication of large RSA numbers (that cannot be factored fast in practice) can still be easily represented as a BDD.

Keywords Binary decision diagrams · Integer factorization · RSA

Mathematics Subject Classification (2010) 11-XX · 94A60

1 Introduction

The integer factorization problem is one of the most basic, yet fascinating and formidable problems in mathematics dating back several centuries. In integer factorization, the input

This article is part of the Topical Collection on *Special Issue: Mathematical Methods for Cryptography*

✉ Håvard Raddum
haavardr@simula.no

Srimathi Varadharajan
srimathi.varadharajan@uib.no

[1] Simula@UiB, Thormøhlensgate 55, 5006 Bergen, Norway

is a positive integer N and the task is to find its prime factors. Given a large N, there is no known efficient algorithm that can factorize N in the classical computational model, although there is a polynomial-time quantum algorithm known as Shor's algorithm [1]. Due to the hardness of the problem, integer factorization forms the basis of the RSA public key cryptosystem. The RSA public key cryptosystem uses two primes p and q, usually of the same bit size. An extensive amount of research has been done in RSA factorization, we refer to the survey papers by Boneh [2] for the complete account.

In this paper we take a completely different approach to the integer factorization problem compared to more well-known methods such as Quadratic sieve [3] and Number Field Sieve [4]. We focus primarily on integers that have only two prime factors p and q of same bit size, although our method can be generalized to numbers with arbitrarily many prime factors. The key point in our approach is to use Binary Decision Diagrams (BDD) [5].

BDDs have been used to represent systems of Boolean equations [6, 7], and we extend this idea to integer factorization. Other researchers have also used BDD-like structures with success in various applications. See for instance [8] for applications to permutations. A BDD is a directed acyclic graph in which there is a root node and every node has at most two outgoing edges. Each edge is labeled either 0 or 1. The nodes in a BDD are arranged in levels where each level corresponds to one variable. The values of the variables are represented by labeled edges (0 or 1) going out from the nodes on some level. We treat the bits in the unknown factors $p = p_{n-1} \dots p_1 p_0$ and $q = q_{n-1} \dots q_1 q_0$ as variables, and equations are obtained by linking the multiplication operation of p and q to the given bits in N.

There are two basic operations we can perform on BDDs: *swapping levels* and *adding levels*. Adding levels results in levels being associated with linear combinations of variables, and not only single variables. Using these two operations we get linear combinations at each level, some of which are linearly dependent. Using the so-called linear absorption algorithm and reduction defined on BDDs [6], we can solve the inherent equation system $N = pq$, and thus find the unknown bits of p and q.

We show that the number of nodes in the initially constructed BDD is $\mathcal{O}(n^3)$ where n is the number of bits in p and q to be precise, we prove the number of nodes is $2n^3 - 2n^2 - 2n + 5 \lesssim B_n \leq 2n^3 - 4n + 5$. Previously, Burch [9] showed that the number of nodes in the BDD for a general multiplier is bounded by $4n^3 - 6n^2 - 4n + 12$ for $n \geq 2$. The method which they use to make a polynomial size BDD is similar to our process by repeating the same variable on several different levels to achieve a smaller BDD. Applying our idea to RSA moduli, we find that the multiplication of large RSA numbers that cannot be factored in practice can still be easily represented as a BDD. We have also run experiments to determine the complexity of the proposed factorization algorithm. The complexity we see is not as good as the best known methods, but our BDD representation is still interesting as a radically different approach to the factorization problem. The main contribution of this work is thus inspiration to look at the problem of integer factorization from a new angle, and in Section 5.2 we show some new aspects of factorization that can be studied using BDDs.

2 Binary decision diagrams

The data structure we use for our approach to the factoring problem is a Binary Decision Diagram (BDD). Binary Decision Diagrams are used in various applications, and it is possible to interpret them in different ways. For a comprehensive treatment of BDDs, see [5].

Here we will give a description of BDDs, with emphasize on our interpretation, visualization, and differences from other definitions. The description is essentially the same as given in [7]. In Fig. 1 we give an example of a BDD, and it may be useful to refer to this figure when reading the description below. In the context of this paper, the variables in the BDD are the bits of the secret primes p and q that multiplies to a known RSA modulus N.

2.1 Description of BDD

A BDD is a particular kind of directed acyclic graph. The nodes in a BDD are arranged in horizontal *levels*, and we visualize a BDD by drawing the levels in a top down fashion. There is only one node on the highest level, called the *top node*, and there is only one node on the lowest level, called the *bottom node*. All edges in the BDD are directed downwards, with an edge always going between nodes on different levels. Each node, except for the bottom node, has one or two outgoing edges, called the *0-edge* and/or the *1-edge*. The bottom node only has incoming edges and no outgoing edges. 0-edges are drawn as dotted lines, while 1-edges are drawn as solid lines.

In other descriptions of BDDs in literature, a level is usually associated with single variable over $GF(2)$ that only occurs on that particular level. Our description differs in two ways. First, we generalize the single variable per level to allow a *linear combination* of variables over $GF(2)$ associated with each level. Second, we allow the same linear combination to appear on different levels or, more generally, we allow dependencies among the linear combinations of the levels. How to handle these linear dependencies is the major part of the algorithm for factoring N.

A *path* in a BDD is a sequence of consecutive edges, where the end node of one edge is the start node for the next edge. A *complete* path starts in the top node and ends in the bottom node. We regard each edge in a path to assign a value to the linear combination associated with the level where the edge starts. If an e-edge starts from a node on a level associated with linear combination l, it yields the linear equation $l = e$ (for $e \in \{0, 1\}$). Thus a complete path gives a system of linear equations, which may or may not have a solution given the unconstrained assignment of linear combinations to the levels. If a path gives a linear system that has a solution we call it a *consistent* path, if not it is an *inconsistent* path.

Note that an edge need not go from one node to a node on the level directly below. A complete path that *jumps* over some levels does not assign values to the linear combinations on the jumped levels.

2.2 Problem solving using BDDs

The power and usefulness of BDDs comes from the fact that a BDD may consist of a small and manageable number of nodes, while having exponentially (in the number of nodes) many paths. On a high level, the actual structure of a given BDD encodes all relations between the variables in the problem we are trying to solve. When these relations follow some structure or pattern, the number of nodes tends to be small even if the problem size is large in some other sense. Any problem represented as a BDD will always be to find values for the variables that satisfy the relations given by the problem. To be more precise, finding the solution to a problem represented as one of our BDDs is always to find a consistent path in the BDD, and solve the associated linear system of equations.

The number of complete paths in a BDD may be very large while having only one or very few consistent paths. We therefore need to have something better than just brute force guessing to find a consistent path. In the next section we explain some operations that can be done on BDDs, and how they can be used to eliminate inconsistent paths without any guessing.

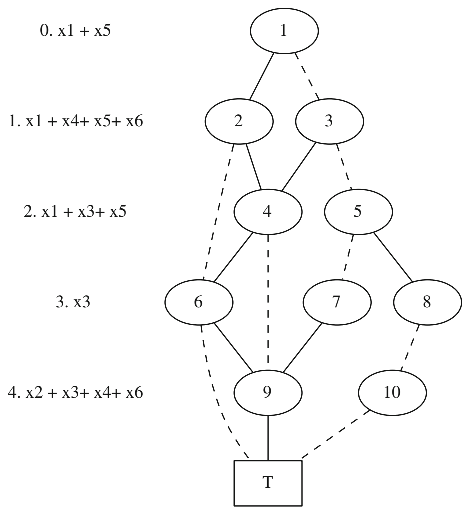


Fig. 1 A binary decision diagram with 6 levels and linear combinations from a set of 6 variables

3 Operations on BDDs

It is possible to delete single nodes or edges in a BDD, but it is not possible to delete single paths. One edge or node is part of a large number of different paths, and we do not know a priori which edges that are part of a, possibly unique, consistent path. The operations described here have all been presented in [6, 10], but we repeat them briefly for completeness and to ease the explanations that follow.

3.1 Swapping levels

When a BDD has been constructed to represent some problem instance, the nodes, edges and levels are fixed, and the linear combination associated with each level is fixed. In [10] Rudell explains how to swap the variables on two adjacent levels in a BDD and change the nodes and edges such that the resulting BDD encodes exactly the same problem instance. The fact that we are using linear combinations at the levels and not single variables is irrelevant here. So we have a method to change the order of the linear combinations associated to the levels, without changing the problem instance or its solution space.

The good thing about the swap operation is that it is a local operation, in the sense that only nodes and edges on the two affected levels need to be changed. The swap operation has linear time complexity in the number of nodes on the two levels, so it is very cheap to do when this number is small. The drawback is that the number of nodes on the lower of the two levels may, in the worst case, double during the operation, so repeatedly doing swap operations through the BDD may lead to exponential growth in the number of nodes. The number of nodes may also decrease during this process, but finding the order of the linear combinations giving the smallest BDD is an NP-hard problem [11].

3.2 Adding levels

In most earlier works the levels have had single variables attached to them. When only single variables on the levels are allowed it does not make sense to add variables together, and ask how the BDD must be altered to keep the problem instance intact. When we have linear combinations associated with the levels, it is natural to want to be able to add them together.

In [6] it is explained how one can add one linear combination for a level onto the linear combination for the level directly below, and change the BDD accordingly to keep the solution space of the problem unchanged. This operation is not as well known as swapping levels, so we explain the general case in more detail here, see Fig. 2. Assume l_1 and l_2 are the linear combinations for two adjacent levels, with l_1 on the highest. If we stand in a node on the l_1-level, then choosing values for l_1 and l_2 will send us to one of the four nodes labeled A, B, C and D in Fig. 2. When we add l_1 to l_2, so the lower level gets associated with $l_1 + l_2$, the same choice of values for l_1 and l_2 must send us to the same node. For instance, $l_1 = 1$ and $l_2 = 0$ leads to node C in the left BDD of Fig. 2. That choice of values gives $l_1 + l_2 = 1$, so $l_1 = 1$ and $l_1 + l_2 = 1$ must also end up in C in the right BDD. In general, to preserve the solution space when replacing l_2 by $l_1 + l_2$ we must flip the outgoing edges for any node pointed to by a 1-edge from the l_1-level.

The procedure for the add operation is similar to the swap operation, and its important properties are the same: Only nodes and edges connected to the two affected levels need to be manipulated, running time is linear in the number of nodes on these levels, but the number of nodes on the lower level may double in the worst case.

With the swap and add operations, we have the tools needed to do Gaussian elimination on the linear combinations of the levels. In particular, any linear combination in the span of the initial ones may be produced to represent one level, while the nodes and edges of the BDD are changed accordingly to keep the underlying problem instance unchanged.

3.3 Linear absorption - removing inconsistent paths

In our definition of a BDD we may have dependencies among the linear combinations on the levels. These dependencies are the source of the inconsistent paths, and removing them gives the solutions of the problem we are trying to solve. Linear absorption [6] identifies and removes inconsistent paths and here we explain briefly how it works. Figure 3 shows a small example of the steps carried out in linear absorption on the BDD in Fig. 1.

Assume $l_0, \ldots, l_r$ are linear combinations associated with some levels in a BDD, and that they are linearly dependent ($l_0 + \ldots + l_r = 0$). Assume furthermore that they appear in order, with l_0 at the highest level among them, and l_r at the lowest level. Some of the paths in the BDD are inconsistent with respect to the constraint $l_0 + \ldots + l_r = 0$ and we would like to remove all such paths.

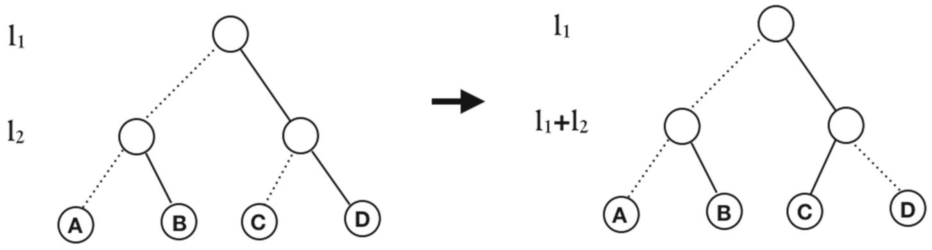
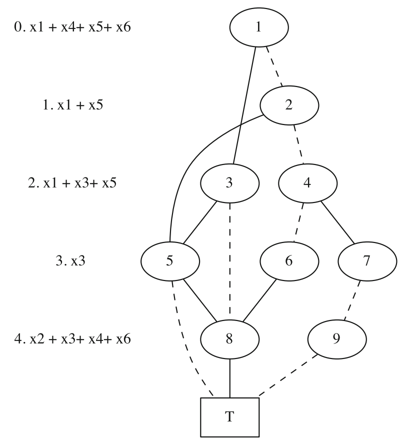
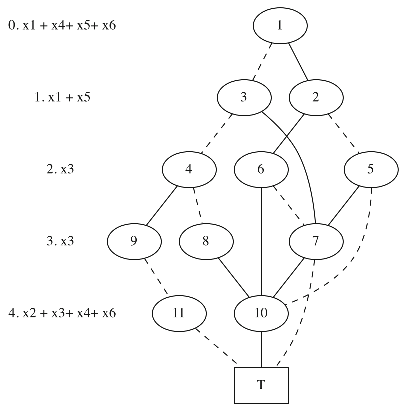


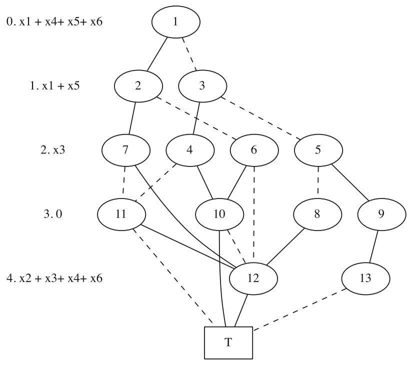
Fig. 2 To preserve the solution space when adding l_1 onto l_2, the outgoing edges of nodes pointed to by a 1-edge from the l_1-level must be swapped



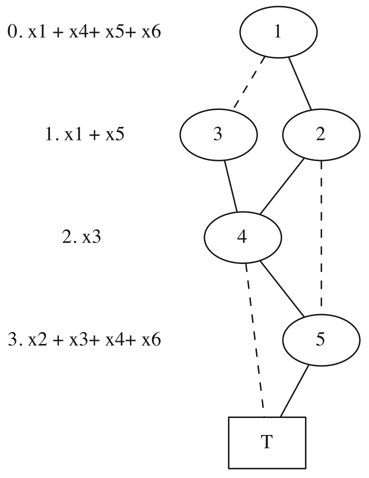
(a) After swapping l_0 and l_1.



(b) After adding l_0 to l_2.



(c) After adding $l_0 + l_2$ to l_3.



(d) After deleting 1-edges and removing 0-level

Fig. 3 Absorbing the linear dependency $l_0 + l_2 + l_3 = 0$ where $l_0 = x_1 + x_5, l_2 = x_1 + x_3 + x_5$ and $l_3 = x_3$. Every BDD in the figure has the same solution space and encodes the same Boolean function

We start with the level with l_0 and repeatedly use the swap operation to move it downwards in the BDD until l_0 is at the level just above l_1. Apply the add operation, changing the level for l_1 into a level associated with $l_0 + l_1$. Use the swap operation again and move $l_0 + l_1$ down to the level just above l_2, then use the add operation to change the level for l_2 into a level for $l_0 + l_1 + l_2$. Continue moving the sum of l_i's downwards, picking up new terms on the way using the add operation.

The final add operation creates the sum $l_0 + \ldots + l_r$ for the level that was associated with l_r. Since the l_i's are linearly dependent we have therefore created a level with $\mathbf{0}$ as its linear combination. The dependency $l_0 + \ldots + l_r = 0$ has now been condensed into a single level, and whether a path is consistent or not with this dependency now only depends on the edges going out from this level. A path having a 1-edge out from the level associated with $\mathbf{0}$ gives a linear system containing the inconsistent "equation" $\mathbf{0} = 1$ directly. Hence all 1-edges going out from the $\mathbf{0}$-level should be removed.

After removing all 1-edges from the $\mathbf{0}$-level, all remaining paths in the BDD will be consistent with the particular dependency we started with, regardless of how we transform the BDD using swap and add operations. When removing the inconsistent 1-edges we say that the linear dependency $l_0 + \ldots + l_r = 0$ has been *absorbed* into the BDD.

A 0-level with only outgoing 0-edges does not give any constraint or information in the quest to find a solution to the problem instance. Any path will now give a system including the equation $\mathbf{0} = 0$, which carries no information. We may therefore remove the whole $\mathbf{0}$-level. This is done by redirecting all edges pointing to nodes on the $\mathbf{0}$-level to their children along the 0-edge, and then deleting all $\mathbf{0}$-level nodes. This decreases the number of nodes as well as reducing the number of levels in the BDD by 1.

3.4 Reducing a BDD

When performing any of the operations described above the BDD may be left in a state where nodes may be merged or deleted. After doing linear absorption it may happen that some internal nodes have no outgoing edges. Such nodes become dead ends for any path reaching it, and will never be part of a complete consistent path. These nodes should naturally be removed. Note that deleting one dead-end node may create other dead end nodes on the level above, and all should be deleted in a recursive fashion.

Linear absorption may also create internal nodes that have no incoming edges pointing to it. These nodes are never part of a complete path and can also safely be removed. Again, deleting one node without incoming edges may create new nodes without incoming edges and a cascade of deletions occurs.

These two cases only happen when linear absorption has been used, deleting 1-edges from a level. However, only doing the swap and add operations may also leave the BDD in a state where nodes can be deleted. This can happen in two ways.

First, it may happen that both the 0- and 1-edges from a node A point to the same node B. It can then be shown that A and B are essentially the same node, and that they can be merged. This is done by redirecting all of A's incoming edges to B, and then deleting A.

Second, if the 0-edges of both A and B point to N_0 and the 1-edges of both A and B both point to N_1 it can be shown that A and B encode the same information, since they have exactly the same sub-graph below them. These nodes can be merged, realized by redirecting B's incoming edges to A and then deleting B.

Reducing the BDD removes nodes and simplifies the BDD as much as possible. We always assume it is being done after executing any of the operations described above. After doing any operation the BDD only needs to be checked for reduction on the affected levels, and recursively up or down the BDD until no more reduction can be done. When no more nodes can be deleted or merged we say that the BDD is *reduced*.

In practice, reduction is a semi-local operation that only affects the nodes close to some particular level and has a linear run time in the number of affected nodes. It has been shown [12] that for a fixed set of linear combinations attached to the levels a reduced BDD is always unique (regardless of the order of deleting nodes).

4 Integer multiplication represented as a BDD

In this section we will show how multiplication of two n-bit numbers can be represented via a BDD. We focus on the case of RSA moduli: $N = pq$ where both $p = (p_{n-1}, \dots, p_0)$ and $q = (q_{n-1}, \dots, q_0)$ have n bits each and N is a fixed and known $2n$-bit value. The straight-forward base 2 multiplication of p and q can be written as follows:

$$\begin{array}{cccccccccccc}
 & (q_{n-1} & \cdots & q_2 & q_1 & q_0) & (p_{n-1} & \cdots & p_2 & p_1 & p_0) \\
\hline
 & & & & & & p_0q_{n-1} & \cdots & p_0q_2 & p_0q_1 & p_0q_0 \\
+ & & & & & p_1q_{n-1} & p_1q_{n-2} & \cdots & p_1q_1 & p_1q_0 & \\
+ & & & & p_2q_{n-1} & p_2q_{n-2} & p_2q_{n-3} & \cdots & p_2q_0 & & \\
\vdots & & & \iddots & \vdots & \vdots & \vdots & \iddots & & & \\
+ & & p_{n-1}q_{n-1} & \cdots & p_{n-1}q_2 & p_{n-1}q_1 & p_{n-1}q_0 & & & & \\
\hline
= & N_{2n-1} & N_{2n-2} & \cdots & N_{n+1} & N_n & N_{n-1} & \cdots & N_2 & N_1 & N_0
\end{array} \tag{1}$$

The $+$-signs are ordinary integer additions. The p_iq_j terms appear in *columns*, and we refer to the terms of p_iq_j where $i + j = c$ as column c. The equations that the p_i and q_j satisfy are given as follows from the rightmost columns:

$$\begin{aligned}
p_0q_0 &= N_0 \\
p_0q_1 + p_1q_0 &= N_1 + v2, \text{ for some } v \text{ (bit-wise carry)} \\
p_0q_2 + p_1q_1 + p_2q_0 &= N_2 + v2^2, \text{ for some } v \text{ (bitwise carry pattern)} \\
&\vdots
\end{aligned}$$

Let v_k be the number given by the k least significant bits of N. Adding up the first k columns of the multiplication above will result in the number $v_k + t2^k$, where t is the carry pattern from the additions that will affect the next columns. Note that the k lsb's of N will be determined by the additions in the first k columns, and will not change when computing the rest of the multiplication.

4.1 Basic BDD building block

The terms that are added when computing the multiplication are of the form p_iq_j, which can have values 0 or 1. The term p_iq_j appears in column $i + j$, and hence will contribute either 0 or 2^{i+j} to N. We can attach the value of N computed *so far* in the nodes in the BDD. If the value computed before processing p_iq_j is v, the value after processing p_iq_j will be v or $v + 2^{i+j}$. This can be captured in the graph structure in Fig. 4. This small subgraph is the basic building block for constructing the BDD representing the whole multiplication $N = pq$.

4.2 Building the multiplication BDD

We build the BDD by going through the multiplication column by column, starting with column 0.

Column 0 Initially the value N computed so far, v_0, is 0. We start the BDD by building the structure for column 0, which only contains the term p_0q_0. The top of the BDD will only contain the basic building block, as shown in Fig. 5a. Since processing p_0q_0 completes the contribution from column 0, the possible values for v_1 are listed in the nodes on the lowest

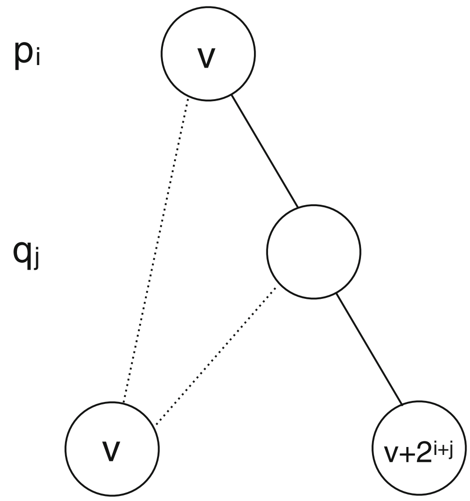


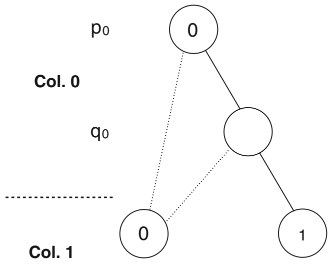
Fig. 4 The term $p_i q_j$ adds 2^{i+j} to v when $p_i = q_j = 1$, and 0 otherwise

level (so far only 0 and 1). We can now check which values that do not match the actual v_1 given by the known N and delete the corresponding nodes. With p and q odd primes, v_1 must be 1 and we delete the node that suggests $v_1 = 0$. The result is shown in Fig. 5b.

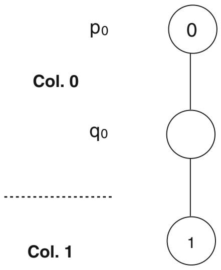
Column 1 We continue building the BDD from the bottom node in Fig. 5b. The first term in column 1 is $p_0 q_1$, so the two new levels we get when adding the basic building block starting from this node will have p_0 and q_1 attached to them. The basic building block has two nodes on the bottom, and the value in these nodes will be 1 in the case that $p_0 q_1 = 0$ and 3 when $p_0 = q_1 = 1$. This is shown in Fig. 5a.

To add the second term $p_1 q_0$ in column 1, we extend the BDD by adding two basic building blocks from each of the bottom nodes we have until now. The levels for the nodes will be associated with p_1 and q_0. Extending from the node containing the value 1 will produce two new bottom nodes with values 1 and 3, respectively. Extending from the node with value 3 will make two new bottom nodes with values 3 and 5. Nodes with equal values can be merged, so we end up with the construction in Fig. 6b.

We have now completed construction for column 1, and can check to see which nodes that have values consistent with the known v_2, the two lsb's of N. If $v_2 = (01)_2$ we delete

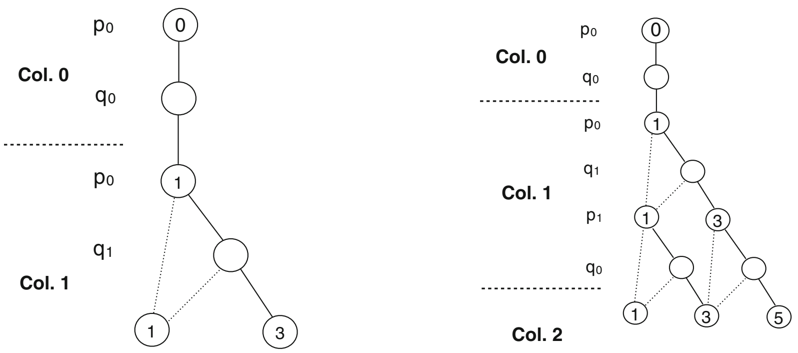


(a) $p_0 q_0$ adds 1 when $p_0 = q_0 = 1$.



(b) After deleting impossible node for v_1.

Fig. 5 BDD after processing column 0



(a) Adding p_0q_1 to the value we have so far.

(b) Complete construction of BDD for columns 0 and 1.

Fig. 6 BDD after processing column 1

the node on the lowest level with value 3, and if $v_2 = (11)_2$ we delete the nodes containing 1 and 5.

Column *k* We can continue building the BDD in a recursive fashion. Assume that we have completed the construction for column $k - 1$ and have a current bottom level with t nodes on it. Let these nodes be $A_0, \ldots, A_{t-1}$, where the value in each A_i is $v_k + i2^k$. We extend a basic building block from each of the A_i's, to add the first term in column k.

The values in the two bottom nodes extending from A_i will be $v_k + i2^k$ and $v_k + (i+1)2^k$. The latter of these values will be the same as the value in the first bottom node extending from A_{i+1}. These nodes can be merged, so all basic building blocks will be linked together as shown in Fig. 7. Due to this linking, the number of nodes on the new bottom level will be $t + 1$ instead of $2t$.

We continue in this fashion for each of the terms in column k. Let the number of terms in column k be a_k. Each term adds one to the number of nodes of the new bottom level, so after adding the basic building blocks for the last term the number of nodes on the bottom level will be $t + a_k$. Their values will be $v_k + i2^k$ for $i = 0, \ldots, t + a_k - 1$. At this point the

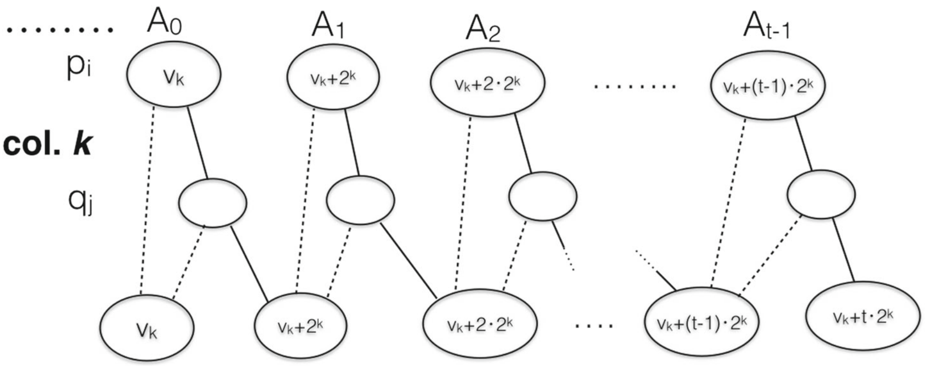


Fig. 7 Adding the first term p_iq_j in column k. Basic building blocks are linked together

addition of column k is complete, and we must check which nodes have values consistent with v_{k+1}. If $N_k = 0$, the nodes with values $v_k + i2^k$ for even i will match the given N, and if $N_k = 1$ the nodes with i odd will be consistent with N. Hence every other node will be inconsistent with v_{k+1} and get deleted, so we end up starting the next column with $\lfloor (t + a_k)/2 \rfloor$ or $\lceil (t + a_k)/2 \rceil$ nodes, depending on the given value of N.

Completing the BDD Once the construction for the final column is done, the full multiplication $pq = N$ is captured in the BDD. At this point only the node on the bottom level having the value N should be kept, and all others deleted. The node with the value N becomes the bottom node of the BDD. Finally, the BDD should be reduced to remove any dead ends remaining.

4.3 Size of constructed BDD

We end the section by proving that the number of nodes in a BDD representing $N = pq$ will contain $\mathcal{O}(n^3)$ nodes. Hence the multiplication of large RSA numbers that can not be factored in practice can still be easily represented as a BDD.

First we give a formula for the number of terms in a particular column. We state this as a lemma for easy reference later. The correctness of the formula is easily seen by inspecting the multiplication table (1) at the start of this section, so we omit a formal proof.

Lemma 4.1 *Let a_k be the number of terms in column k of the multiplication $pq = N$ where p and q are two n-bit numbers. Then*

$$a_k = \begin{cases} k+1, & 0 \le k \le n-1 \\ 2n-k-1, & n \le k \le 2n-2 \end{cases}$$

We continue by counting the number of nodes on the bottom level when starting the construction of the nodes for column k. Our aim is to get an upper bound on the number of nodes in the BDD. In the following, when we have t nodes on the bottom level before deleting nodes according to the known v_k we will always assume that the number of nodes remaining after deletion is $\lceil t/2 \rceil$.

Lemma 4.2 *Let t_k be the number of nodes on the level where nodes representing the addition of the terms in column k starts. Then*

$$t_k = \begin{cases} 1, & k = 0 \\ k, & 1 \le k \le n \\ 2n-k+1, & n+1 \le k \le 2n-1 \end{cases}$$

Proof The case for $k = 0$ is special. The BDD starts with a single top node, so $t_0 = 1$ is true by definition. We prove the case $1 \le k \le n-1$ by induction. The statement is true for $k = 1$ since we always start with a single node after column 0 is complete when making nodes for column 1. Assume that $t_l = l$. The addition of the first term in column l will add two new levels to the BDD, the first will contain l nodes and the second will contain $l+1$ nodes. Each addition of a new term adds two new levels, and the number of nodes on the second of these will have one more node than the previous. Since there are $a_l = l+1$ terms in column l, the number of nodes on the bottom level after adding the last term will be $l + (l+1) = 2l+1$. After deleting half of the nodes inconsistent with the given v_{l+1}, we get the number for nodes starting column $l+1$ as $t_{l+1} = \lceil (2l+1)/2 \rceil = l+1$. This shows the correctness for the case $1 \le k \le n$.

We show the formula for $n+1 \le k \le 2n-1$ also by induction, but the base case is less trivial this time. We know that $t_n = n$ and need to show that $t_{n+1} = 2n-(n+1)+1 = n$ to start the induction. The construction of nodes for column n starts with t_n nodes. In a similar fashion as explained above, two new levels gets added for each term in column n and the second of these increases by one node from the previous. The bottom level after adding all a_n terms is the starting level for column $n+1$, and will contain $t_n + a_n$ nodes. We then delete half of these nodes according to the value of v_{n+1}. With $t_n = n$ and $a_n = n-1$ from Lemma 4.1 we get $t_{n+1} = \lceil (2n-1)/2 \rceil = n$, verifying the base case.

Assume now $n+1 \le l \le 2n-2$ and $t_l = 2n-l+1$. As explained above the number of nodes on the level starting column $l+1$ will be $t_l + a_l$ before deleting nodes, and $\lceil (t_l + a_l)/2 \rceil$ after. We then get $t_{l+1} = \lceil (2n-l+1+2n-l-1)/2 \rceil = \lceil (4n-2l)/2 \rceil = 2n-(l+1)+1$, as desired. □

With a_k and t_k defined for $0 \le k \le 2n-2$ we can count the (maximum) number of nodes in the part of the BDD representing additions in column k.

Lemma 4.3 *Let T_k be the number of nodes in the BDD representing additions in column k. Then*

$$T_k = \begin{cases} 2, & k = 0 \\ 3k^2 + 3k, & 1 \le k \le n-1 \\ 3n^2 - 5n + 2, & k = n \\ 3(2n-k-1)(2n-k), & n+1 \le k \le 2n-2 \end{cases}$$

Proof By construction from previous columns, the level starting additions in column k contains t_k nodes. Each term in column k adds first a new level with the same number of nodes as the level above, and then a level with one node more. This continues for each of the a_k terms in column k, so the last two levels added have $t_k + a_k - 1$ nodes and $t_k + a_k$ nodes, respectively. The last level belongs to column $k+1$ and should not be counted, so we get $t_k + t_k + (t_k+1) + (t_k+1) + \ldots + (t_k + a_k - 1) + (t_k + a_k - 1)$ nodes for column k in total. This can be written as

$$T_k = \sum_{i=0}^{a_k-1} 2(t_k + i) = 2a_k t_k + 2 \sum_{i=0}^{a_k-1} i = 2a_k t_k + a_k(a_k - 1) = a_k(2t_k + a_k - 1).$$

It is now straight forward to verify the four cases stated in the lemma by inserting the expressions for a_k and t_k from Lemmas 4.1 and 4.2 and do the calculations. □

We are now ready for the main result.

Theorem 4.4 *Let $N = pq$, where N is known and p and q are two unknown n-bit numbers. Let B_n be the number of nodes in the BDD representing $N = pq$. Then*

$$B_n \le 2n^3 - 4n + 5.$$

Proof The proof follows by just summing up the values for T_k from Lemma 4.3 for all the columns, and adding 1 for the bottom node. In the calculations we make use of the standard formulas

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \text{ and } \sum_{k=1}^{n} k = \frac{n(n+1)}{2}.$$

We complete the proof by showing some of the calculations:

$$\begin{aligned}
B_n &\le 1 + \sum_{k=0}^{2n-2} T_k = 1 + T_0 + \sum_{k=1}^{n-1} T_k + T_n + \sum_{k=n+1}^{2n-2} T_k \\
&= 1 + 2 + \sum_{k=1}^{n-1}(3k^2 + 3k) + (3n^2 - 5n + 2) + \sum_{k=n+1}^{2n-2} 3(2n-k-1)(2n-k) \\
&= 3n^2 - 5n + 5 + 3\sum_{k=1}^{n-1} k^2 + 3\sum_{k=1}^{n-1} k + 3\sum_{k=1}^{n-2}(2n-(n+k)-1)(2n-(n+k)) \\
&= n^3 + 3n^2 - \frac{13}{2}n + 5 + 3\sum_{k=2}^{n-1}(k-1)k \\
&= 2n^3 - 4n + 5
\end{aligned}$$

□

We can estimate a lower bound on the number of nodes in the BDD by always rounding downwards when half of the nodes on the level starting a column are deleted. We then get the following values for t_k:

$$t_k = \begin{cases} 1, & 0 \le k \le 1 \\ k-1, & 2 \le k \le n \\ 2n-k, & n+1 \le k \le 2n-1 \end{cases}$$

The values for a_k remain the same. Re-doing the calculations for T_k and B_n with these values gives us rather tight bounds on the number of nodes in the BDD representing $pq = N$. However, the actual number of nodes in the final BDD might actually be somewhat smaller than the estimated lower bound, due to the final reduction step after completing construction for all columns. For this reason the lower bound is not exact, but the main point that the number of nodes in the BDD is approximately $2n^3$ remains.

Corollary 4.5 *Let $N = pq$, where N is known and p and q are two unknown n-bit numbers. Let B_n be the number of nodes in the BDD representing $N = pq$. Then*

$$2n^3 - 2n^2 - 2n + 5 \lesssim B_n \le 2n^3 - 4n + 5.$$

5 Factoring experiments and other observations

We have constructed many BDDs for various values of N, following the description given in the previous section. For two n-bit numbers $p = (p_{n-1} \dots p_0)$ and $q = (q_{n-1} \dots q_0)$ there will be n^2 different terms $p_i q_j$ going into computing the multiplication pq. Each term will give two levels in the constructed BDD, so the total number of levels in the BDD will be $2n^2$. There are exactly $2n$ unknown variables p_i and q_j, and each variable initially occurs on exactly n different levels. See Fig. 8 for an example of what the structure of a complete multiplication BDD looks like. Note the pattern that emerges from the borders between different columns, where half the nodes have been deleted.

Each path in the BDD suggests values for the unknown p_i's and q_j's. Almost all of the paths are inconsistent, because a path may very well choose $p_i = 0$ on one level where p_i

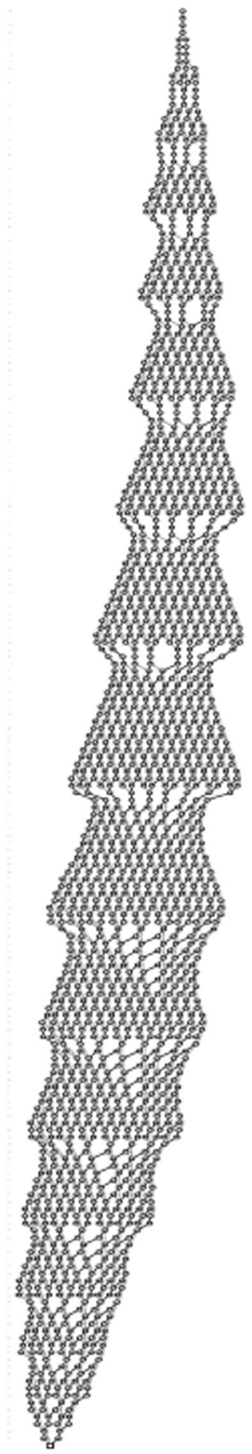


Fig. 8 BDD for $N = 471953$

occurs and $p_i = 1$ on another. If we can remove all inconsistent paths, any remaining path in the BDD will give the values of the p_i's and q_j's such that $pq = N$.

5.1 Observed complexity of factoring

We have done experiments using linear absorption to remove inconsistent paths. There are $2n$ variables in total and $2n^2$ levels, so there will be $2n^2 - 2n$ different dependencies we need to absorb before all dependencies are gone and every remaining path is consistent. Once we have reached that point there will only be two paths remaining in the BDD. Both of them give values for p and q such that $pq = N$ (there are two paths because it is undecided which factor is p and which is q, i.e. for $N = 77$ we can have $p = 7, q = 11$ or $p = 11, q = 7$).

An RSA modulus represented as a BDD can be factored using linear absorption, but we need a measure of its complexity. As proved in Theorem 4.4, the initial BDD will contain only polynomially many nodes. When doing the swap and add operations during linear absorption the number of nodes will start to grow. However, the final BDD after all dependencies have been absorbed is very small, since it only contains two paths. At some point the BDD will therefore reach a maximum number of nodes, and operating on this BDD will give the heaviest work, both in time and space. Hence we take the maximum number of nodes during factoring as our measure of complexity.

It is very hard to predict in advance how many nodes the BDD will contain after absorbing a number of dependencies. Hence we do not have a closed formula $f(n)$ for the complexity of factoring an n-bit number using our method. There are various heuristics and strategies one can employ when it comes to which order we should absorb the linear dependencies. Some are slightly better than others, but in all our experiments the big picture is that the complexity of factoring $2n$-bit numbers with the BDD approach and linear absorption is of the order 2^n. Table 1 shows the details and actual runtimes for some particular values of N. The experiments were done on a MacBook Air with a 1.3 GHz Intel Core i5 processor and 8GB RAM.

5.2 Some observations on the multiplication BDD

As far as we know, factoring RSA moduli via the method described in this paper can not compete with the best factoring algorithms known (i.e. Number Field Sieve) in terms of asymptotic complexity. The purpose of this paper is rather to give a different approach to the factoring problem, and maybe inspire some new ideas. Below we present two observations that might be useful for future work.

Table 1 Some factoring experiments

N	p	q	$\lceil \log_2(N) \rceil$	Peak number of nodes	Runtime (seconds)
479069	571	839	19	$2^{12.996}$	0.316
1887239	1249	1511	21	$2^{14.070}$	0.760
8795869	2741	3209	24	$2^{14.925}$	1.246
288676361	16603	17387	29	$2^{18.347}$	18.86
9657443137	93407	103391	34	$2^{20.136}$	80.3
163580897747	405917	402991	38	$2^{22.303}$	537

5.2.1 Absorbing $2n-2$ dependencies for free

Finding an order to absorb the dependencies such that the overall complexity is minimized is an unsolved problem. However, from the starting BDD it is possible to absorb $2n-2$ dependencies without any increase in the number of nodes. This can be explained as follows:

The terms in column $k(< n)$ are $p_0q_k, p_1q_{k-1}, \ldots, p_kq_0$, but the order of adding the terms is insignificant when computing N. In the BDD the variables p_i and q_{k-i} must appear on adjacent levels, but the (p_i, q_{k-i})-pairs can be put in any order on the levels representing column k without changing the BDD. Also, which one of p_i and q_{k-i} that appears first in the (p_i, q_{k-i})-pair is arbitrary. Permuting the variables like this within one column does not change the BDD at all.

Say now that p_i appears in both column k and column $k-1$. It is then possible to put the (p_i, q_{k-i-1})-pair at the lowest possible levels within column $k-1$, and set p_i as the lowest of these two. In column k we can set the (p_i, q_{k-i})-pair at the highest possible levels within column k, and p_i at the very highest. The two levels with p_i will now be adjacent and can be merged into one, absorbing one dependency. The number of nodes in the BDD

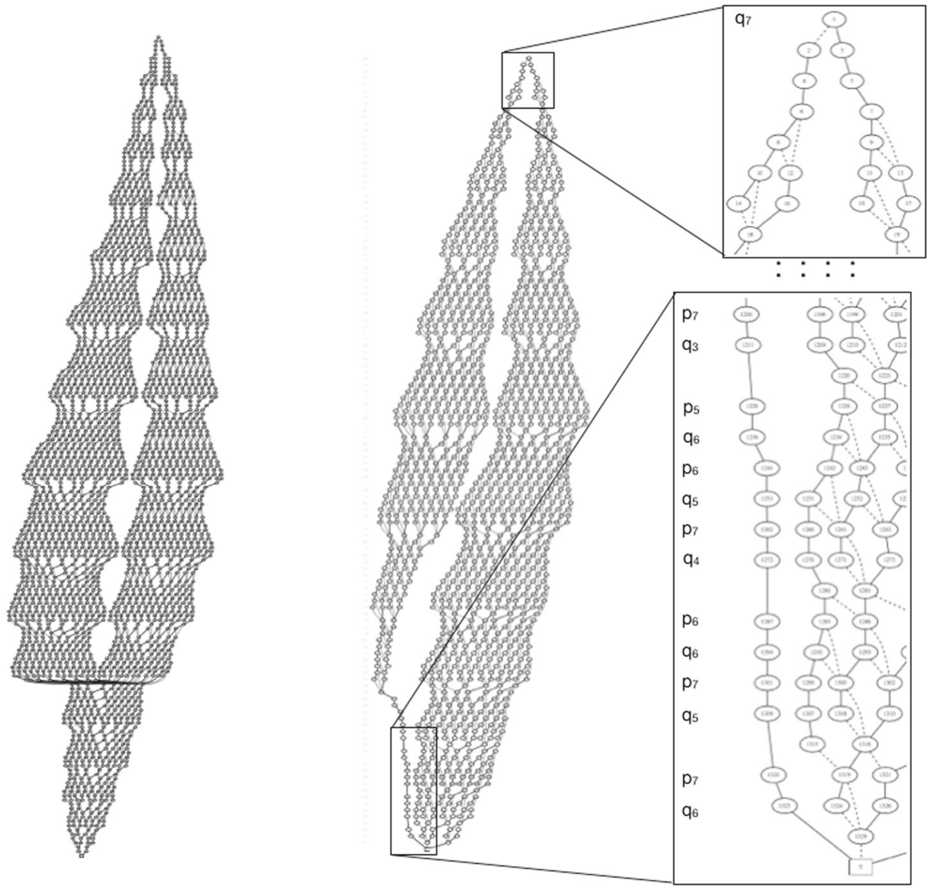


Fig. 9 Absorbing all instances of one variable and moving it to the top splits the BDD in two

will actually decrease since the BDD was unchanged up until the two p_i-levels were added together, and then the lower of the two levels was deleted.

We can let two variables meet like this on each of the $2n - 2$ column boundaries, and absorb $2n - 2$ variables without any increase in the number of nodes. Unfortunately, this process can not be readily repeated since the column boundaries now consist of three levels that are dependent on each other and can not freely move without increasing the number of nodes. To merge other variables that are equal means that one of them has to cross a column boundary, with no guarantees on the resulting number of nodes.

5.2.2 *Moving one variable to the top*

Each p_i- and q_j-variable initially occurs on n different levels. Let us start with the lowest level where, say, p_i is found, and use the swap operation to move it upwards through the BDD. Each time this instance of p_i is just below another level with p_i, we merge the levels with linear absorption before continuing. In the end, the variable p_i is only found at the highest level, where the top node sits.

The resulting BDD has now been split into two parts between the level where we started; one part for $p_i = 0$ and one for $p_i = 1$ (see left BDD in Fig. 9). If it is possible to somehow detect which part that contains the consistent path we would learn whether $p_i = 0$ or $p_i = 1$, delete the wrong part of the BDD by cutting off the 1- or 0-edge from the top node, and iterate the process with another variable.

In the right BDD in Fig. 9 we show an example where we have moved q_7, initially found at the lowest level, to the top. The BDD splits completely in two, and in this example we can see that guessing $q_7 = 0$ immediately determines the values of several other variables. All paths in the part of the BDD where $q_7 = 0$ ends in the same string of 1-edges, indicating that $p_5 = p_6 = p_7 = q_3 = q_4 = q_5 = q_6 = 1$ must be true for there to be any possibility for p and q to multiply to N.

6 Conclusion

In this paper, we have shown how to use Binary decision diagrams to factor a number N which has two factors p and q. Unfortunately, for the factoring experiments of RSA moduli of $2n$ bits, the running time and space complexity seems to be of order 2^n which is not better than prevalent methods. The main take away of our approach is that we can store the information about the factors of N in the form of a BDD of size polynomial in n, $O(n^3)$ to be precise.

This paper gives a different light on how to factor a number N, in the sense that we keep and store all the intermediate multiplications of individual bits in the form of a BDD. This is a very different approach compared to the existing more number theoretic methods for factorization. It allows us to study the details of what really goes on inside the multiplication process itself, by representing N via the multiplications of bits from the unknown factors.

References

1. Leighton, F.T., Shor, P.W. (eds.): Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4–6, 1997. ACM, New York (1997)
2. Boneh, D.: Twenty years of attacks on the RSA cryptosystem. Not. AMS **46**, 203–213 (1999)

3. Pomerance, C.: The quadratic sieve factoring algorithm. In: Advances in Cryptology: Proceedings of EUROCRYPT 84, A, Workshop on the Theory and Application of of Cryptographic Techniques, Paris, France, April 9–11, 1984, Proceedings, pp. 169–182 (1984). https://doi.org/10.1007/3-540-39757-4_17
4. Lenstra, A.K., et al.: The number field sieve. In: Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13–17, 1990, Baltimore, Maryland, USA, pp. 564–572 (1990). https://doi.org/10.1145/100216.100295. http://doi.acm.org/10.1145/100216.100295.,
5. Knuth, D.E.: The Art of Computer Programming, vol. 4. Addison-Wesley Professional (2009)
6. Schilling, T.E., Raddum, H.: Solving compressed right hand side equation systems with linear absorption. In: Sequences and Their Applications - SETA 2012 - 7th International Conference, Waterloo, ON, Canada, June 4–8, 2012. Proceedings, pp. 291–302 (2012). https://doi.org/10.1007/978-3-642-30615-0_27
7. Raddum, H., Kazymyrov, O.: Algebraic attacks using binary decision diagrams. In: Cryptography and Information Security in the Balkans - First International Conference, BalkanCryptSec 2014, Istanbul, Turkey, October 16–17, 2014, Revised Selected Papers, pp. 40–54 (2014). https://doi.org/10.1007/978-3-319-21356-9_4
8. Minato, S.: πDD: a new decision diagram for efficient problem solving in permutation space. In: Theory and Applications of Satisfiability Testing - SAT, 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19–22, 2011. Proceedings, pp. 90–104 (2011). https://doi.org/10.1007/978-3-642-21581-0_9
9. Burch, J.R.: Using BDDs to verify multipliers. In: Proceedings of the 28th Design Automation Conference, San Francisco, California, USA, June 17–21, 1991, pp. 408–412 (1991). https://doi.org/10.1145/127601.127703. http://doi.acm.org/10.1145/100216.100295.
10. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, 1993, Santa Clara, California, USA, November 7–11, 1993, pp. 42–47 (1993). https://doi.org/10.1109/ICCAD.1993.580029
11. Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Trans. Computers **45**(9), 993–1002 (1996). https://doi.org/10.1109/12.537122
12. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers **35**(8), 677–691 (1986). https://doi.org/10.1109/TC.1986.1676819