Safe BDD Minimization Using Don't Cares*

Youpyo Hong Peter A. Beerel
Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA 90089

Jerry R. Burch Kenneth L. McMillan
Cadence Berkeley Laboratories
Berkeley, CA 94704

Abstract

In many computer-aided design tools, binary decision diagrams (BDDs) are used to represent Boolean functions. To increase the efficiency and capability of these tools, many algorithms have been developed to reduce the size of BDDs. This paper presents heuristic algorithms that minimize the size of BDDs representing incompletely specified functions by intelligently assigning don't cares to binary values. The traditional algorithm, restrict [8], is often effective in BDD minimization, but can increase the BDD size. We propose new algorithms based on restrict which are guaranteed never to increase the size of the BDD, thereby significantly reducing peak memory requirements. Experimental results show that our techniques typically yield significantly smaller BDDs than restrict.

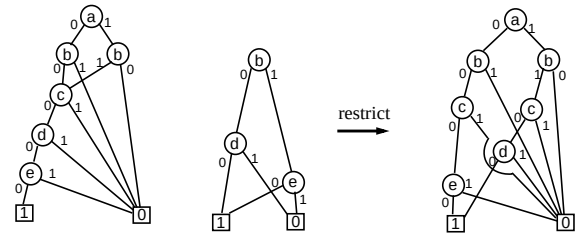
1 Introduction

The efficient representation and manipulation of Boolean functions is critical to many computer-aided design applications including logic synthesis, formal verification, and testing. Binary decision diagrams (BDDs) have proven to be an efficient means of representing and manipulating many commonly used Boolean functions. For BDD-based tools, the size of the BDDs can determine their run-time efficiency, the problem size that they can handle, and/or the quality of the circuits they synthesize [5]. Many techniques have been developed to find BDD variable orderings that lead to compact BDDs [1, 2]. For a given variable ordering, the BDD representation of a completely specified function is unique. For incompletely specified functions, however, many BDDs can be used to represent the function, each associated with a different assignment of don't cares (DCs) to binary values. This paper assumes the variable ordering is fixed and addresses the problem of finding an assignment of DCs that yields a small BDD representation.

Finding the smallest BDD using *don't cares* is known to be NP-complete [3] and exact techniques [4] are not applicable to most practical applications. Therefore, heuristic algorithms have been developed to address this *BDD minimization* problem. These heuristics try to maximize the instances of *node sharing* and *sibling-substitution* [12] during the minimization process. BDD nodes become *shared* if the re-assignment of DCs makes their associated sub-functions identical. Sibling-substitution is a special case of node sharing where a child of a BDD node *u* is replaced by the other child of *u*, thereby removing *u* and the child.

Chang *et al*. [11] proposed a method to share multiple nodes at each level in the BDD in top-down order. The reduction potential of their method is large, but its high computational complexity prohibits its application to large BDDs.

The *restrict* operator and the *constrain* operator (also known as *generalized-cofactor)* [8, 9, 10] are well known BDD minimization algorithms. Both algorithms implement sibling-substitution recursively. Shiple *et al.* [12] proposed a framework to relate these heuristics and explored several variants. Their experimental results suggest that the *restrict* operator and its variants are efficient in terms of both run-time and resulting BDD size. A common problem of these techniques, however, is that the size of the BDD may increase as a result of the 'minimization', as illustrated in Figure 1. An obvious way to avoid using a larger BDD is to compare the original BDD with the 'minimized' BDD and use the smaller one. We refer to this approach as *thresholding.* The potential for size increase, however, suggests that these methods may not produce BDDs as small as those produced by algorithms that guarantee that no sub-BDD becomes larger.



(a) BDD to minimize (b) Care-BDD (c) Result

Figure 1: Example of restrict increasing BDD size.

This paper describes *safe* BDD minimization heuristics, *i.e*., they guarantee the resulting BDD is not larger than the original BDD. These algorithms are based on the observation that *restrict*, while always reducing the size of the target sub-BDD, can increase the size of a BDD which contains the target sub-BDD. Consider the left child *d* of node *c* in the BDD depicted in Figure 1 (a). Node *d* can be reached from the root by two different *paths* and has two different associated care sub-sets, represented by node *d* and *e* in the Care-BDD depicted in Figure 1 (b). In *restrict*, sibling-substitution is applied to the right child of *d* (replacing it with *e*) because the right child of the Care-BDD node *d* corresponds to a DC, resulting in a smaller sub-BDD rooted at *c*. However, since a different sub-BDD rooted at *c* is needed for the other path, node *c* becomes unshared. We call this effect *node splitting*. This splitting of *c* leads to the overall size increase illustrated in Figure 1 (c).

We develop *safe* minimization heuristics by performing sibling-substitution only on the nodes which we can guarantee will not cause an overall increase in BDD size. These techniques can lead to better minimization results by preventing sibling-substitutions that cause growth while allowing sibling-substitutions elsewhere. Compared to *thresholded restrict* our algorithms produced up to 27% smaller BDDs. The results compared to *restrict* are much more dramatic, because *restrict* increased the size of many BDDs we tested.

After defining the problem and presenting relevant notations in Section 2, we present our two techniques called *basic compaction*

*This work was supported in part by a gift from Intel and a NSF CAREER Award MIP-9502386.

34th Design Automation Conference

DAC 97, Anaheim, California

and *leaf-identifying compaction* in Section 3. Section 4 reports our experimental results and we present some conclusions in Section 5.

2 Preliminaries

We denote the set of nodes of a BDD [7] with N and the set of edges with E. There are two types of nodes in N, leaf nodes and non-leaf nodes. The leaf nodes are either 0 or 1, representing the Boolean functions 0 and 1, respectively. Each non-leaf node *u* has two outgoing edges; a *then_edge* and an *else_edge.* Each edge is connected to a *child* node of *u* and *u* is the *parent* of the child nodes. The two child nodes are *siblings* of each other. Each non-leaf node F is associated with a Boolean variable *x*. The child of F reached via the then_edge is denoted by F_x; the other child is denoted by $F_{\bar{x}}$. When not ambiguous, we will use F to refer both to a function and a BDD that represents the function. The size of a BDD F, denoted $|F|$, is the number of nodes in F.

The domain of any single-output Boolean function ff can be partitioned into three subsets, ff_{on} (the on-set), ff_{off} (the off-set), and ff_{dc} (the don't care set). A completely specified function has ff_{dc} empty, while an incompletely specified function has a non-empty don't care set. Any two of these sets uniquely describes an incompletely specified function.

An incompletely specified function ff can be represented by a pair of completely specified functions $[f, c]$ for which $f_{on} \supseteq ff_{on}$, $f_{off} \supseteq ff_{off}$, and $c = ff_{on} \cup ff_{off}$. For a given incompletely specified function ff, there are many such functions f, each referred to as a cover of ff, representing different partitions of ff_{dc} into f_{on} and f_{off}.

Definition 1 Given an incompletely specified function ff, f' is a **cover** of ff if $f'_{on} \supseteq ff_{on}$, and $f'_{off} \supseteq ff_{off}$.

In practice, [f,c] is typically given in the form of a BDD pair [F, C]. Among all covers of ff, there must be at least one cover f' whose BDD F′ is smallest in size. Unfortunately, finding a smallest F′ for a given [F,C] pair representing ff is NP-complete [3], so we consider heuristic approaches. Given [F, C], finding an F′ that is hopefully close to minimal in size is called *BDD minimization using don't cares*. We call F the original BDD and F′ the *minimized* BDD, even though traditional algorithms can yield an F′ larger than F. (The term *reduced* BDD is not used because it may be misinterpreted as a result of the reduction procedure used in generating BDDs.)

3 Minimization Algorithms

This section develops heuristic BDD minimization algorithms that guarantee the minimized BDD is no larger than the original. Our algorithms consist of two phases. In the first phase, the original BDD is preprocessed to identify nodes for which applying sibling-substitution does *not* increase overall BDD size. In the second phase, sibling-substitution is selectively applied to the nodes identified in the first phase.

We present two algorithms. The first one, called *basic compaction*, is a simple technique that performs a subset of the sibling-substitutions that do not cause *node-splitting*. The second one, called *leaf-identifying compaction*, which is more sophisticated, effectively allows node-splitting in the special case that the new node is a leaf.

3.1 Basic Compaction

To explain our basic approach we define a one-to-many map between edges in F and nodes in C. Specifically, we say the then_edge (else_edge) of a node *u* in F *maps* to all nodes *v* in C for which there exists a partial input combination *p* and an extended version *p′* with $x = 1$ ($x = 0$) such that F evaluates to *u* for *p* and to u_x ($u_{\bar{x}}$) for *p′* and C evaluates to *v* for *p′*, where *x* is the variable associated with node *u*.

Consider first the case in which each edge in F maps to exactly one node in C. This occurs when no non-leaf nodes in F are shared (have more than one parent) as in the BDDs depicted in Figure 2. For this simple case, we can easily predict the structure of the minimized BDD as a result of sibling-substitution as follows. If an edge between nodes *v* and *u* in F matches to the 0 leaf in C, this means that the extended partial input combination represents a don't care set in the original function. Consequently, we can remove *u* and all nodes below *u* and replace *u*'s parent with its sibling. If *u* matches to a 1 leaf (care leaf) in C, then *u* and its descendants must be preserved in the minimized result. Lastly, if *u* matches to a non-leaf node, *u* will be preserved in the minimized result even though its descendants may or may not be preserved (depending on whether or not they match to a don't care). In other words, we can predict which nodes can be removed by analyzing the mapping between edges in F and nodes in C. In our basic algorithm, we mark an edge in F if the child node connected to it is to be preserved, as illustrated in Figure 2.

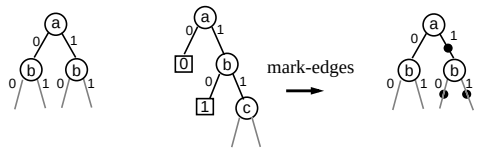


Figure 2: Identifying necessary nodes: no F nodes shared

Now, consider a general BDD in which nodes may be shared. Then, an edge between nodes *v* and *u* in F can map to multiple nodes in C. If any of these nodes is not a DC, we can conservatively assume that substituting *u* with *u*'s sibling may cause node-splitting (i.e., node *u* or a modified version of it is needed). Consequently, we mark all incoming *then_edges* (*else_edges*) for a node if there is any corresponding node in C that is not a DC. For example, in Figure 3, the *else_edge* of *c* in F matches with both the 0 and 1 leaves in C and therefore it is marked. In contrast, *restrict* performs sibling substitution if any mapped node in C is a DC.

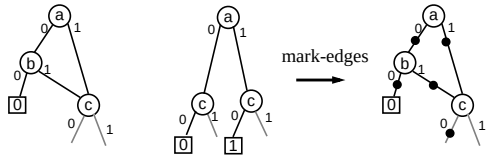


Figure 3: Identifying necessary nodes with shared nodes

After the first phase, called *mark-edges*, is completed, the second phase, we call *build-result*, rebuilds the BDD F solely based on the markings on edges in F. If an edge from a node *v* to any of its child nodes *u* is not marked, then *v* can be safely replaced by *u*'s sibling via sibling-substitution. Otherwise, *v* is preserved and its children are recursively rebuilt. Figure 4 illustrates this *selective* sibling-substitution-based rebuilding technique on an edge-marked BDD.

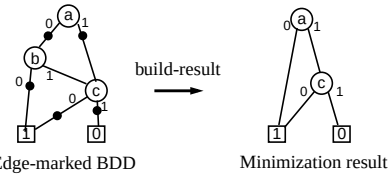
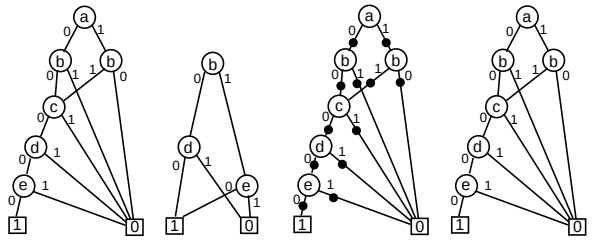


Figure 4: Minimizing BDD by using marking result

Figure 5 illustrates the results of both phases of the algorithm on the example given in Figure 1. Notice that all edges are marked and the result is the same BDD as the original one. That is, this algorithm, referred to as *basic compaction*, prevents the BDD from



(a) BDD to minimize (b) Care-BDD (c) Edge-marked BDD (d) Result

Figure 5: An example of basic compaction

growing.

Figure 6 presents the pseudo-code of *basic compaction*. The time complexity of *mark-edges* is $O(|F|\cdot|C|)$ because each pair of nodes from F and C is called only once by using an operation cache. Due to the application of another operation cache, *build-result* processes each node only once, yielding a time complexity of $O(|F|)$. *Clear-edges* routine clears the edge-marking fields after building the result and has time complexity of $O(|F|)$. Consequently, the overall time complexity of *basic compaction* is $O(|F|\cdot|C|)$, the same complexity as *restrict*.

Basic compaction is a *correct* BDD minimization, as defined below (proof omitted).

Definition 2 A BDD minimization using don't cares is correct if the minimized BDD is a cover of the original BDD.

Theorem 1 *Basic compaction* is correct.

Now, we show that *basic compaction* is *safe***.** Recall that a BDD minimization using don't cares is safe if the minimized BDD is guaranteed to be no larger than the original BDD, i.e. $|F| \geq |F'|$.

Theorem 2 *Basic compaction* is safe.

Proof: The result of *basic compaction* is produced by *build-result.* Recall that *build-result* takes F as its only argument and that the nodes of F have two Boolean labels: a then_edge and an else_edge. For the purposes of this proof, it does not matter how these labels are set by *mark-edges*. Inspection of *build-result* shows it does not affect the status of any then_edge or else_edge. Thus, for a given sub-BDD G of F, a call to *build-result* with argument G always produces the same result. Let F′ be the result of *build-result* applied to F. It is easy to show by induction on the depth of F that every sub-BDD of F′ (including F′ itself) is the result of calling *build-result* on some sub-BDD of F. Thus, since *build-result* always produces the same result when applied to a given sub-BDD of F, the number of sub-BDDs of F′ is no larger than the number of sub-BDDs of F. Thus, the size of the F′ is no larger than the size of the F. So, we can conclude that *basic compaction* is safe. ❑

Intuitively, *basic compaction* is safe because it ensures that no nodes will be split. This property can be deduced from the structure of *build-result*. It creates one node for each node it visits (which uniquely depends on the edge-marking) and visits each node at most once (because of the operation cache). Specifically, nodes that are not reachable from the root by a path of marked edges are not visited by *build-result* and thus not included in the minimized BDD.

3.2 Leaf-identifying Compaction

This subsection presents an enhanced safe minimization technique in which a special type of node splitting is allowed. Consider the set of sibling-substitutions applied to a child *u* with respect to its parent *v*. When the results of all the substitutions for *u* are unique, then the sibling-substitutions can increase the BDD size only by the size of the unique result. Leaf nodes are special in that they are essential for all non-trivial BDDs. So, the idea of new algorithm is to accept the result of sibling-substitution if the result is a unique leaf (*i.e.* replace the edge from *v* to *u* with an edge from *v* to the leaf). Note that, *u* may be preserved or replaced in the minimized

```
bdd basic-compaction (bdd f, bdd c) {
   if (c == bdd_ zero) return (bdd_zero);
   mark-edges(f, c);
   result = build-result(f);
   clear-edges(f);
   return(result);
}
void mark-edges (bdd f, bdd c) {
   if (c == bdd_ zero) return;
   if (f == leaf) return;
   x = top variable(f, c);
   if (c_x != bdd_zero)
      if (f != f_x) f.then_mark = 1;
      mark-edges(f_x, c_x);
   if (c_x' != bdd_zero)
      if (f != f_x') f.else_mark  = 1;
      mark-edges(f_x', c_x');
}
bdd build-result(bdd f) {
   if (f  == leaf) return(f);
   x = top variable(f);
   if (f.then_mark == 1 and f.else_mark == 0)
      return (build-result(f_x));
   else if (f.then_mark  == 0 and f.else_mark == 1)
      return (build-result(f_x'));
   else
      return (bdd_find(x, build-result(f_x), build-result(f_x')));
}
```

Figure 6: Basic compaction pseudocode

BDD if it has multiple parents, depending on sibling-substitutions with respect to its other parents.

This approach will usually lead to better results for two reasons. First, a sub-BDD can be replaced by a leaf which might be preserved in *basic compaction*. We call this type of gain as *Gain 1*. Second, the number of edges marked can be less than in *basic compaction* because the edge-marking routine needs not recur through edges to be redirected to leaves. This type of gain is called *Gain 2*. Typically, less edge-markings leads to smaller BDDs because *build-result* removes nodes connected by unmarked edges. Note, however, that this approach is not guaranteed to produce better results than *basic compaction* because the two algorithms can result in different unshared nodes becoming shared unpredictably.

This new approach can be implemented using a two-phase edge-marking routine and a modified *build-result*. The first phase of edge-marking computes the results of all possible sibling-substitutions from which it identifies the edges that can be redirected to leaves. The second phase is similar to basic edge-marking except that it does not recur through edges that can be redirected to leaves. After edge marking, the modified *build-result* routine redirects all identified edges to their annotated leaf and applies sibling substitution to all remaining unmarked edges.

Figure 7 shows an example (that is slightly different from the one in Figure 5) where both gains contribute in minimizing the BDD. First, the *then_edge* from node *a* and the *then_edge* from the node *b* on the right can be redirected to the 0 leaf (Gain 1). Consequently the *then_edge* of node *d* is unmarked (Gain 2). The modified *build-result* routine leads to a minimized BDD with two nodes less than the original BDD. In contrast, *basic compaction* leads to no minimization because basic edge-marking must mark all edges.

The time complexity of this approach is almost twice as much as *basic compaction* because of the two-phase edge-marking routine since each edge-marking phase requires $O(|F|\cdot|C|)$. If we do not pursue the gain from fewer marked edges (*Gain 2*), it is possible to merge the two phases of edge-marking into one. Our experiments suggest that degradation of quality is negligible. We believe this is because it is unlikely that all nodes on the paths leading to

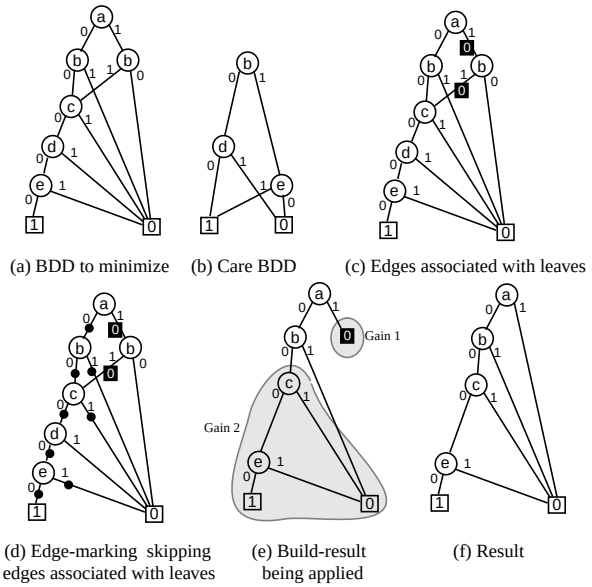


Figure 7: Improved result by leaf-identification

an excessively marked edge can be redirected to a unique leaf (so that no marking is required for the edge). Thus, this comprise represents a good performance/run-time trade-off.

We refer to this enhanced algorithm with the above compromise as *leaf-identifying compaction* and it is given in Figure 9. Finding and annotating nodes is performed in a preprocessing phase called *LI-mark-edges*. Like *restrict*, this phase recursively performs sibling-substitution. However, instead of returning the actual BDD result, it returns a classification of the result. This classification identifies whether the edge can be redirected to a 1 (encoded b01), 0 (encoded b10), DC (encoded b00), or non-leaf (encoded b11). The encoding facilitates a bitwise-OR scheme that implements the relative priority of non-leaves over leaves and leaves over DCs. Figure 8 illustrates an example of *leaf-identifying compaction* where one edge, the *then_edge* of *d*, is additionally marked compared to the example in Figure 7 (d).

The overall time complexity of *leaf-identifying compaction* is the same as the complexity of *basic compaction* which is $O(|F|\cdot|C|)$.

4 Experimental Results

We conducted experiments to compare our two heuristics to *restrict*, thresholded *restrict*, *osm_bt* [12] and *thresholded osm_bt*. *Osm_bt* was chosen among a variety of heuristics developed by Shiple *et al*. because it showed the best overall results in the examples they tested. The heuristics were incorporated into a formal verification tool VIS [13] and were tested on BDDs found during symbolic reachability analysis [14]. All experiments were run on an Ultra SPARC 1/192M.

In our first experiment, monolithic transition relation BDDs (TR BDDs) were minimized using two types of DCs. The first DC set is the state transitions whose next states are already known to be reachable (*i.e.*, reached states.) In particular, state transitions with such a next state can be removed/added to the TR BDD with no effect on the final reachable state set [13]. The second DC set is derived from the *frontier set*, *i.e.*, the set of states newly found to be reachable by the previous iteration of reachability analysis. In particular, state transitions in the TR BDD whose present state is not in the *frontier-set* are don't cares [13]. After minimization, the next states is computed by first conjuncting the TR and frontier-set and then existentially quantifying out the present state variables. Note that the frontier-set based don't cares must be recalculated in each iteration based on the new frontier set found.

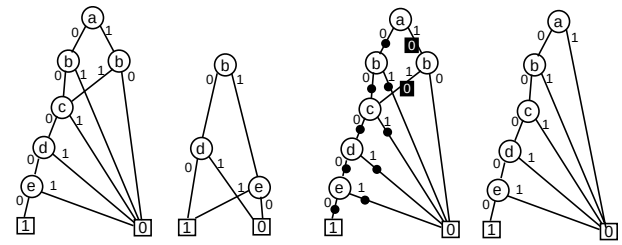


Figure 8: Leaf-identifying compaction

```
bdd LI-compaction (bdd f, bdd c) {
    if (c == bdd_ zero) return (bdd_zero);
    (void) LI-mark-edges(f, c);
    result = LI-build-result(f);
    clear-edges(f);
    return(result);
}

int LI-mark-edges (bdd f, bdd c) {
    if (c == bdd_zero) return (00);
    if (f  == bdd_one)  return (01);
    if (f  == bdd_zero) return (10);
    x = top variable(f, c);
    temp1  = LI-mark-edges(f_x, c_x);
    temp2  = LI-mark-edges(f_x', c_x');
    if (f != f_x)
        f.then_mark =  f.then_mark | temp1;  /* '|' is  bitwise-or */
        f.else_mark  = f.else_mark |  temp2;
    return (temp1 | temp2);
}

bdd LI-build-result(bdd f) {
    if (f  == leaf) return(f);
    x = top variable(f);
    if (f.then_mark == 11) f_left = LI-build-result(f_x);
    else if (f.then_mark == 01) f_left = bdd_one;
    else f_left = bdd_zero;

    if (f.else_mark == 11) f_right = LI-build-result(f_x');
    else if (f.else_mark == 01) f_right = bdd_one;
    else f_right = bdd_zero;

    if (f.then_mark == 00 and f.else_mark != 00) return f_right;
    else if (f.then_mark != 00 and f.else_mark == 00) return f_left;
    else return (bdd_find(x, f_left, f_right));
}
```

Figure 9: Leaf-identifying compaction pseudocode

The minimization results on the TR BDDs from the VIS-1.1 example circuits are given in Table 1. We experimented with two DC sets separately. Each DC set varies from iteration to iteration. For each iteration we calculate the *DC fraction*, which is the percentage of input combinations for which the function value is a DC. We report its range over all iterations for each example. The average BDD minimization results over all iterations for *restrict*, *thresholded restrict*, *osm_bt*, *thresholded osm_bt*, *basic compaction*, and *leaf-identifying compaction* are listed in columns denoted *R*, *TR*, *O*, *TO*, *B*, and *L*, respectively.

The results show that *restrict* and *osm_bt* typically increase the BDD size if the DC fraction is very small--less than 1%. On the other hand, as expected, both *compaction* routines never increase BDD size. Of all four heuristics, *leaf-identifying compaction* generates the smallest BDDs in most cases.

Our second experiment tested BDDs that represent the combi-

national logic which makes up the finite state machine representation of each circuit. Once the reachability state analysis is finished, the unreachable states can be set to DCs for subsequent tasks, *e.g.* synthesis. We report the DC fraction for each combinational logic block associated with a primary output. This fraction is obtained from the Care-BDD after all non-supporting variables of the target BDD are existentially quantified out. Table 2 summarizes our results.

When each example is given equal weight, *thresholded osm_bt* shows slightly better result than both *compaction*s. If each example is weighted by the number of nodes in its TR BDD, however, *leaf-identifying compaction* yields better results in general. This suggests that *leaf-identifying compaction* is more effective when the BDD size is large. To justify this trend more explicitly, Figure 10 illustrates the BDD minimization ratio vs. original BDD sizes. We see that *leaf-identifying compaction* outperforms *thresholded osm_bt* for all BDDs whose size is larger than 150. We believe this is because node-splitting occurs more often in larger BDDs due to a higher degree of node sharing.

We also explored the relationship between minimization ratio and DC fraction. Our results in Figure 11 clearly illustrates a positive correlation between BDD minimization ratio and DC fraction.

Our *compaction* routines successfully completed on all examples but were sometimes significantly slower than *restrict*. *Leaf-identifying compaction* shows lower runtimes than *basic compaction* when *leaf-compaction* yields a smaller result than *basic compaction*. We believe that this is because *leaf-identifying compaction* needs to visit less nodes than *basic compaction* in building a result. Both *compactions* are significantly faster than *osm_bt* and *thresholded osm_bt*, which do not complete within two hours on the four largest examples we tested. Thus, our heuristics demonstrate a good run-time performance/reduction quality tradeoff.

5 Conclusion

We describe two low-complexity heuristics for BDD minimization using don't cares that guarantee non-increasing BDD sizes and yield significantly smaller BDDs than obtained with the traditional algorithm, *restrict*. These heuristics use edge-marking techniques to inhibit the growth of sub-BDDs while still allowing some sub-BDDs to shrink, thereby having lower peak memory usage than *restrict*. This can be a significant advantage in memory-bounded applications. In addition, the proposed BDD minimization methods may lead to significant run-time improvements for a variety of applications, especially where the run-time of minimization can be amortized over many operations involving the minimized BDD. Our experimental results also demonstrate a positive correlation between BDD minimization quality and DC fraction. This suggests that a simple, yet powerful heuristics to reduce run-times may be to selectively run BDD minimization depending on the DC fraction.

Exploring the impact of BDD minimization using don't cares in logic synthesis [6] would also be interesting future work.

Acknowledgments

The authors would like to thank Dr. Rolf Drechsler of the Albert-Ludwigs-University Freiburg for valuable discussions and feedback on an earlier version of this work.

References

[1] H. Fujii, G. Ootomo, and C. Hori, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams," Proc. International Conference on Computer-Aided Design, pp.38-41, 1993.

[2] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," Proc. International Conference on Computer-Aided Design, pp.42-47, 1993.

[3] M. Sauerhoff and I. Wegener, "On the Complexity of Minimizing the OBDD Size for Incompletely Specified Functions," IEEE Trans. Computer-Aided Design, vol. 15, pp. 1435-1437, Nov. 1996.

[4] A. L. Oliveira, L. Carloni, T. Villa and A. Sangiovannir-Vincentelli, "Exact Minimization of Boolean Decision Diagrams Using Implicit Techniques," Technical Report UCB ERL M96/16, University of California, 1996.

[5] L. Lavagno, P. McGeer, A. Saldanha and A. L. Sangiovanni-Vincentelli, "Timed Shannon Circuits: A powerful Design Style and Synthesis Tool," Proc. Design Automation Conference, pp. 254-260, 1995.

[6] M. Damiani and G. De Micheli, "Don't Care Set Specifications in Combinational and Synchronous Logic Circuits," IEEE Trans. Computer-Aided Design, vol. 12, pp. 365-388, March. 1993.

[7] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE Trans. Computers, vol. C-35, pp. 677-691, 1986.

[8] O. Coudert, C. Berthet and J. C. Madre, "Verification of Synchronous Sequential Machines Based on Symbolic Execution," Automatic Verification Methods for Finite State systems, Springer-Verlag, pp. 365-373, 1989.

[9] O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," Proc. International Conference on Computer-Aided Design, pp.126-129, 1990.

[10] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's," Proc. International Conference on Computer-Aided Design, pp.130 - 133, 1990.

[11] S. Chang, D. I. Cheng and M. Marek-Sadowska, "Minimizing ROBDD Size of Incompletely Specified Multiple Output Functions," Proc. the European Design and Test Conference, pp. 620-624, 1994.

[12] T. Shiple, R. Hojati, A. Sangiovanni-Vincentelli, and R. K. Brayton, "Heuristic Minimization of BDDs Using Don't Cares," Proc. Design Automation Conference, pp. 225-231, 1994.

[13] R. K. Brayton and et al., "VIS: A System for Verification and Synthesis," Technical Report UCB/ERL M95, University of California, Berkeley, Dec, 1995.

[14] J. R. Burch, E. M. Clarke, D. Long, K. L. McMillan and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," in Proc. Design Automation Conference, pp. 46-51, 1990.

Circuits	|F|	Iter	DCs from reached states: DC fraction (%)	Avg. |F'|/|F| (%): R	TR	O	TO	B	L	DCs from non-frontier states: DC fraction (%)	Avg. |F'|/|F| (%): R	TR	O	TO	B	L
ping_pong	23	3	6.25-31.25	7.25	8.7	4.35	8.7	1.45	**10.14**	6.25-31.25	**63.77**	**63.77**	**63.77**	**63.77**	**63.77**	**63.77**
tlc	77	8	1.56-37.5	18.67	18.67	19.81	19.81	10.06	**19.48**	81-98.43	**84.09**	**84.09**	**84.09**	**84.09**	**84.09**	**84.09**
ctlp3	138	8	0.78-14.84	-6.07	0	-8.06	0	0	**2.54**	98.43-99.22	**90.22**	**90.22**	**90.22**	**90.22**	89.86	90.04
crd	151	4	0.78-16.79	-16.23	0	-16.06	0	0	**0.83**	91.4-99.21	68.38	68.38	68.54	68.54	68.21	**69.54**
exampleS	292	11	0.05-1.02	-35.31	0	-36.46	0	0	**0.56**	99.85-99.95	94.08	94.08	94.08	94.08	**94.21**	94.18
emodel	343	7	0.39-23.34	11.95	12.79	16.41	17.2	2.5	**21.07**	75.59-92.13	68.05	68.05	69.26	69.26	70.51	**76.51**
dcnew	641	12	0-4.43	-172.28	0	-175.52	0	0	0	99.98-99.9	-23.69	19.55	-20.28	19.6	45.2	**47.14**
gigamax	789	8	0.0015-0.48	-57.78	0	-57.81	0	0	0	99.8-99.9	74.38	74.38	74.56	74.56	72.94	**76.2**
bakery	1155	77	0-0.42	-172.57	0	-194.87	0	0	0	99.9-100	75.24	75.24	75.24	75.24	74.29	**76.58**
abp	1262	22	0.003-2.78	-143.91	0	-147.94	0	0	0	99.86-99.9	82.99	82.99	83.16	83.16	81.11	**84.31**
arbiter	1948	8	0.006-0.7	-32.19	0	-32.31	0	0	0	99.97-99.99	96.24	96.24	96.24	96.24	96.04	96.18
eisenberg	1972	42	0-0.001	-67.86	0	-102.55	0	0	0	96.15-97.9	83.21	83.21	83.24	83.24	79.7	**84.35**
tcp	6835	3	1.56-12.84	-55.07	0	timeout	timeout	2.56	**7.45**	92-98.44	24.53	24.83	timeout	timeout	32.45	**33.81**
amp	9604	164	10e-8	-9.08	0	timeout	timeout	0	0	99.9-100	**99.54**	**99.54**	timeout	timeout	**99.54**	**99.54**
elevator	24640	27	2e-7-0.02	-94.32	0	timeout	timeout	0	0	99.9-100	88.09	88.09	timeout	timeout	86.29	**92.96**
scheduler	73145	38	2.3e-8-0.01	-37.9	0	timeout	timeout	0	0	99.9-100	**95.5**	**95.5**	timeout	timeout	95.06	95.33
Total : equal weight for circuits			N/A	-53.92	2.51			1.04	3.88	N/A	72.79	75.51			77.08	79.03
Total : weighted by BDD size			N/A	-51.22	0.049			0.16	0.51	N/A	89	89.24			89.08	90.83

Table 1: Minimization results on monolithic transition relation BDD. (**|F|**: original BDD Size, **Iter**: reachability analysis iterations, **DC fraction**: range of DC fraction, **Avg. |F'|/|F|**: average minimization ratio, **timeout**: longer than 2 hours, **R**: restrict, **TR**: thresholded restrict, **O**: osm_bt, **TO**: thresholded osm_bt, **B**: basic compaction, **L**: leaf-identifying compaction. Best results are bold-faced if non zero.)

Circuits	Σ|F|	Num. F	Avg. |F|	DCs from reached states: DC fraction (%)	Avg. |F'|/|F| (%): R	TR	O	TO	B	L
ping_pong	19	3	6.33	25-62.5	5.26	**21.05**	5.26	**21.05**	**21.05**	**21.05**
tlc	76	3	25.33	62.5	**27.63**	**27.63**	26.67	27.63	25	25
ctlp3	184	4	46	12.5-75	**29.35**	**29.35**	35.33	35.33	13.59	14.67
crd	79	5	15.8	12.5-62.5	3.8	7.59	8.86	**11.39**	7.59	7.59
exampleS	109	6	18.17	37.5-97.3	39.45	39.45	**47.71**	**47.71**	38.53	38.53
emodel	96	5	19.2	37.5-76.66	7.29	8.33	**17.71**	**17.71**	1.04	6.25
dcnew	172	6	28.67	50-91.13	-2.33	5.81	2.33	7.56	**9.88**	**9.88**
gigamax	2568	10	256.8	78.13-97.46	10.01	10.32	10.71	11.02	**17.17**	**17.17**
bakery	345	12	28.75	37.5-92.11	**19.71**	**19.71**	**19.71**	**19.71**	17.39	17.97
abp	674	8	84.25	25-80.66	3.12	5.19	5.93	6.23	10.83	**26.41**
arbiter	192	16	12	25-93.75	**26.56**	**26.56**	**26.56**	**26.56**	22.4	22.4
eisenberg	408	7	58.29	56.25-70.31	16.91	16.91	**19.61**	**19.61**	6.13	8.09
tcp	652	10	65.2	1.56-84.51	21.47	**25**	21.47	22.24	23.31	**23.31**
amp	509	42	12.12	50-99.87	63.85	63.85	65.94	**66.21**	63.85	63.85
elevator	640	28	22.86	37.5-98.8	27.19	30.47	30.0	**34.84**	20.94	23.75
scheduler	477	20	23.85	37.5-78.51	**12.58**	**12.58**	**12.58**	**12.58**	6.08	6.5
Total : circuit with equal weight				N/A	19.49	21.86	22.27	24.21	19.05	20.78
Total : circuit weighted by BDD size				N/A	13.67	14.9	15.55	16.29	14.31	16.5

Table 2: Minimization results on BDDs for combinational logic cones (**∑|F|**: total BDD size; **Num. F**: number of BDDs, other notations are the same as in Table 1. Best results are bold-faced if non-zero.)

Circuits	|F|	Iter	DCs from reached states: DC fraction (%)	Avg. |C|	Avg. Run-times (msec): R	B	L	DCs from non-frontier states: DC fraction (%)	Avg. |C|	Avg. Run-times (msec): R	B	L
tcp	6835	3	1.56-12.84	73.7	56.7	170	160	92-98.44	61.33	36.7	110	106.7
amp	9604	164	10e-8	192.9	3.48	186.2	184.6	99.9-100	44	4.5	57.6	53.5
elevator	24640	27	2e-7-0.02	1906	307.8	1841.1	1814	99.9-100	777.4	130.7	333	300
scheduler	73145	38	2.3e-8-0.01	1114.2	1378.7	7094	7462.4	99.9-100	469.5	130.5	783.7	827.9

Table 3: Runtime comparisons between minimization heuristics. (Examples from the 4 largest monolithic transition relation BDDs. **Avg. |C|**: average size of the Ca BDD, other notations are the same as in Table 1.)

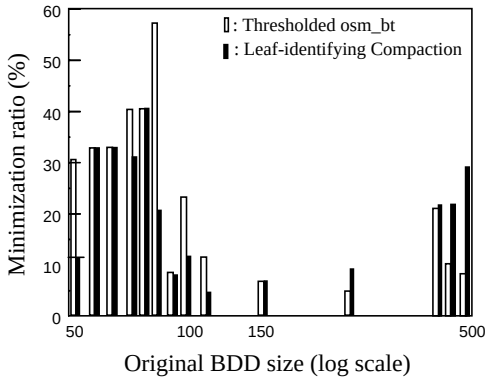


Figure 10: Minimization ratio vs. original BDD size. (Examples from combinational logic BDDs. BDDs smaller than 50 are not included. Close data points are represented using one bar by their average.)

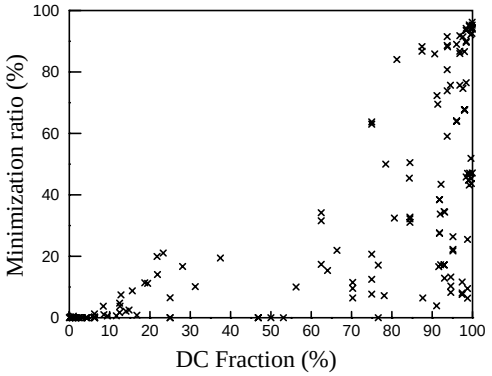


Figure 11: Minimization ratio vs. DC fraction in leaf-identifying compaction. (BDDs smaller than 20 are not included.)