Variable Reordering in Binary Decision Diagrams

Chuan Jiang, Junaid Babar, Gianfranco Ciardo, Andrew S. Miner and Benjamin Smith
Department of Computer Science, Iowa State University
Ames, IA 50011, USA
Email: {cjiang, junaid, ciardo, asminer, bensmith}@iastate.edu

***Abstract*—The size of a BDD heavily depends on its variable order. Significant efforts have been made to find good variable orders, statically or dynamically. This paper concentrates on a related issue, transforming a BDD from one variable order to another, as needed when an application must cope with BDDs having different variable orders. Instead of rebuilding BDDs as done in previous work, we accomplish such transformation through a sequence of adjacent variable swaps. Since there are many ways to schedule these swaps, we propose and compare several heuristics to determine good schedules.**

I. Introduction

A binary decision diagram (BDD) [6] is a data structure to represent and manipulate Boolean functions, with many important real–world applications such as hardware verification and model checking [8]. Since the choice of variable order can significantly impact the size of a BDD (thus its performance) and since finding an optimal variable order is NP-hard [5], significant efforts have been made towards good ordering heuristics. Static heuristics [1], [12], [18] establish a variable order using information available prior to building any BDD, while dynamic heuristics [4], [20] attempt to improve the variable order by modifying the existing BDDs on the fly.

We consider a related topic, *variable reordering*, i.e., how to transform an existing BDD from one variable order to another. This naturally arises when transferring BDDs between applications that choose different variable orders, or when a single application has different computational phases with different optimal variable orders. For example, in reachability analysis with a disjunctively-partitioned next-state function $\mathcal{N} = \bigcup \mathcal{N}_\alpha$, each $\mathcal{N}_\alpha$ could benefit from a possibly different variable order; then, before applying $\mathcal{N}_\alpha$, the already-discovered set of states should be reordered according to $\mathcal{N}_\alpha$'s variable order. Another example arises when two BDDs have been separately improved using dynamic heuristics: before using them in a binary operation, we must compute a common order and transform them into that order efficiently [10].

Several works have proposed variable reordering algorithms based on rebuilding the BDD with the new variable order. Tani and Imai [22] build an equivalent quasi-reduced BDD first, then apply reduction rules on it. Bern, Meinel, and Slobodová [3] adopt a top-down recursive approach and use compute tables to avoid recomputation. Savickỳ and Wegener [21] build the new BDD in breadth first order, but require reverse edges (from a child to its parent) to find a backward path, a feature not usually found in BDD implementations.

Instead, we obtain the desired variable reordering through a sequence of adjacent variable swaps. Relying on swaps is not new (e.g. CUDD [**?**] and BuDDy [**?**] accomplish variable reordering using our BU heuristic of Section III). However, to the best of our knowledge, this approach has not been investigated or compared with rebuilding in the literature, especially in regard to heuristics to schedule swaps.

The rest of this paper is organized as follows: Section II gives basic BDD definitions and reviews the algorithm of adjacent variable swap. Section III analyzes the minimum number of swaps needed for a reordering, and proposes heuristics to choose among minimum-length swap schedules. Section IV presents experimental results for combinatorial circuits and Petri net reachability sets. Section V contains conclusions and directions for further research.

II. Preliminaries

A. Binary decision diagrams

Given domain variables $\mathcal{D} = \{v_1, ..., v_L\}$ and an order Π on $\mathcal{D}$, where $v_i \prec_\Pi v_j$ means "v_i precedes v_j in Π", an *(ordered) binary decision diagram* (BDD) with order Π is an acyclic directed edge-labeled graph where:

- The only *terminal* nodes are the elements of $\mathbb{B} = \{\mathbf{0}, \mathbf{1}\}$, associated with a variable v_0, s.t. $v_0 \prec_\Pi v_i$ for any $v_i \in \mathcal{D}$.
- A *nonterminal* node p is associated with a domain variable, denoted $p.var$, and has two outgoing edges labeled 0 and 1, pointing to *children* denoted $p[0]$ and $p[1]$, s.t. $p[0].var \prec_\Pi p.var$ and $p[1].var \prec_\Pi p.var$.

The function $f_p : \mathbb{B}^L \to \mathbb{B}$ encoded by a node p is given by

$$f_p(i_1, ..., i_L) = \begin{cases} p & \text{if } p.var = v_0 \\ f_{p[i_k]}(i_1, ..., i_L) & \text{if } p.var = v_k \in \mathcal{D} \end{cases}.$$

A node p is said to be *redundant* if $p[0] = p[1]$, while two nodes p and q are said to be *duplicates* if $p.var = q.var$, $p[0] = q[0]$, and $p[1] = q[1]$. To ensure that each function has a unique BDD node representing it, we restrict ourselves to *canonical* forms of BDDs containing no redundant nodes and no duplicates. A non-canonical BDD can be transformed into this *reduced form* [6] but, in practice, one can avoid creating redundant and duplicate nodes during BDD manipulations by employing a *unique table* (usually a hash table) [7].

Some BDD definitions assume a single *root* node (with no incoming edges), thus focus on the encoding of a particular function, and use the term *shared* BDD for our definition, which has no such requirement. We assume that, given a BDD as we defined, there is a set of nodes (which includes all roots but possibly also internal nodes), corresponding to the set $\mathcal{F}$ of

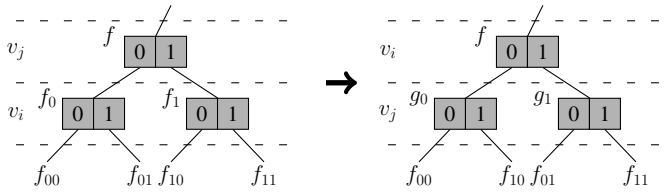

Fig. 1. Swapping v_i and v_j in f.

functions of interest. The other nodes in the BDD also encode a set of functions, $\mathcal{G}$ (where $\mathcal{F} \cap \mathcal{G} = \emptyset$ and $|\mathcal{F} \cup \mathcal{G}|$ is the number of nodes in the BDD), but these functions are not of interest. This is important because, if we reorder the BDD variables, the functions of interest encoded by the original BDD must be preserved in the new BDD, whose nodes will then encode the disjoint sets of functions $\mathcal{F}$ and $\mathcal{G}'$. $\mathcal{G}$ and $\mathcal{G}'$ might not share many (or even any) elements and, of course, the new order results in a smaller BDD iff $|\mathcal{G}'| < |\mathcal{G}|$.

A BDD node can encode a set $\mathcal{Y} \subseteq \mathbb{B}^L$ through its *characteristic function*, i.e., $f_p(i_1, ..., i_L) = 1 \Leftrightarrow (i_1, ..., i_L) \in \mathcal{Y}$. The size of a BDD is not directly related to the size of the encoded set (for example, any set requires as many nodes as its complement), but can be very sensitive to the variable order.

B. Swapping adjacent variables

We can swap adjacent variables v_i and v_j, with $v_i \prec_\Pi v_j$, in time proportional to the number of nodes labeled by v_j [14], without affecting, or even accessing, the nodes below v_i or above v_j, as long as the unique table is partitioned into subtables, one for each variable, and the nodes can be modified *in place*.

Let f be the function associated with a node labeled by v_j. If we express f using the *Shannon expansion*:

$$f = v_j f_1 + \bar{v}_j f_0 = v_j(v_i f_{11} + \bar{v}_i f_{10}) + \bar{v}_j(v_i f_{01} + \bar{v}_i f_{00}),$$

swapping v_i and v_j corresponds to rearranging these terms as:

$$f = v_i(v_j f_{11} + \bar{v}_j f_{01}) + \bar{v}_i(v_j f_{10} + \bar{v}_j f_{00}) = v_i g_1 + \bar{v}_i g_0.$$

Thus, the node encoding f must be relabeled and overwritten with new content: its variable changes from v_j to v_i, and its children change from the nodes encoding f_0 and f_1 (which are disconnected) to the nodes encoding g_0 and g_1 (which must be created). The nodes encoding f_{11}, f_{10}, f_{01}, and f_{00} are reused without change. Fig. 1 illustrates such rearrangement. This algorithm is fundamental to dynamic variable ordering techniques (sifting [20] or its variations [15], [16], [19]).

III. Variable reordering with swaps

Since swapping adjacent variables can be done efficiently, we use a sequence of swaps to transform a BDD from one variable order to another. Let B be a BDD over $\mathcal{D} = \{v_1, ..., v_L\}$. We write $B \sim \Pi$ if B has variable order Π. A *schedule* specifies a sequence of pairs of adjacent variables to swap. The BDD variable reordering problem we consider is:

> Given a BDD B_s, with $B_s \sim \Pi_s$, encoding a set $\mathcal{F}$ of functions of interest, and a variable order Π_t, find a schedule of swaps to transform B_s into B_t, where $B_t \sim \Pi_t$ and B_t encodes all functions in $\mathcal{F}$.

This section analyzes the number of swaps required for this transformation and proposes heuristics for scheduling swaps.

A. Inversions between two orders

Let Π_t be our target order, Π_n the current order of the variables in $\mathcal{D}$, and Π_{n+1} an order resulting from swapping two adjacent elements in Π_n. We say that v_i and v_j form an *inversion* if their relative orders are different in Π_n and Π_t, i.e., if $v_i \prec_{\Pi_n} v_j$ and $v_j \prec_{\Pi_t} v_i$, or $v_j \prec_{\Pi_n} v_i$ and $v_i \prec_{\Pi_t} v_j$. We call this inversion *swappable* if v_i and v_j are adjacent in Π_n. Let $I(\Pi_n, \Pi_t)$ denote the total number of inversions between Π_n and Π_t:

$$I(\Pi_n, \Pi_t) = \sum_{1 \le i,j \le L, v_i \prec_{\Pi_n} v_j} I_{i,j}(\Pi_n, \Pi_t), \qquad (1)$$

where $I_{i,j}(\Pi_n,\Pi_t) = 1$ if v_i and v_j form an inversion, 0 if not. $I(\Pi_n, \Pi_t)$ is the *Kendall tau distance* [17], the number of swaps bubble-sort requires to sort a list of items [**?**].

Theorem 1: It is always possible to transform Π_n into Π_t using exactly $I(\Pi_n, \Pi_t)$ swaps of adjacent elements.

Proof: Omitted (easily proved by induction). ■

Let $S(\Pi_n, \Pi_t)$ denote the number of swappable inversions between Π_n and Π_t:

$$S(\Pi_n, \Pi_t) = \sum_{1 \le i,j \le L, v_i \preccurlyeq_{\Pi_n} v_j} I_{i,j}(\Pi_n, \Pi_t) \qquad (2)$$

where $v_i \preccurlyeq_{\Pi_n} v_j$ means that v_i immediately precedes v_j in Π_n. It is easy to see that $S(\Pi_n, \Pi_t)$ is always less than or equal to $I(\Pi_n, \Pi_t)$, and at most $|\mathcal{D}| - 1$.

Theorem 2: Given an order Π_n and a target order Π_t, let Π_{n+1} be the order obtained from Π_n by swapping a swappable inversion in Π_n. Then, $S(\Pi_{n+1}, \Pi_t)$ can only equal $S(\Pi_n, \Pi_t) - 1$, $S(\Pi_n, \Pi_t)$, or $S(\Pi_n, \Pi_t) + 1$.

Proof: Consider four adjacent variables in Π_n, satisfying $v_i \preccurlyeq_{\Pi_n} v_j \preccurlyeq_{\Pi_n} v_k \preccurlyeq_{\Pi_n} v_l$, where v_j and v_k form an inversion, i.e., $v_k \prec_{\Pi_t} v_j$. If we swap v_j and v_k, the order becomes $v_i \preccurlyeq_{\Pi_{n+1}} v_k \preccurlyeq_{\Pi_{n+1}} v_j \preccurlyeq_{\Pi_{n+1}} v_l$ and, as the swap does not induce changes below v_i or above v_l, the number of swappable inversions not involving v_j or v_k remains the same. Then, it suffices to consider all possible relative orders of these four variables in Π_t, subject to $v_k \prec_{\Pi_t} v_j$, and verify that the number of swappable inversions involving v_j or v_k can only decrease by one, remain the same, or increase by one. Table I lists the $4!/2 = 12$ cases in lexicographic order and shows the swappable inversions involving v_j or v_k in Π_n and Π_{n+1}, and the net change $S(\Pi_{n+1}, \Pi_t) - S(\Pi_n, \Pi_t)$, for each case. The situation where v_j is the lowest or v_k the highest variable in Π_n is analogous and omitted. ■

Applying Theorem 1 to the BDD variable reordering problem, we write the BDD sequence resulting from a schedule of swaps as $B_s \equiv B_0 \to B_1 \to \cdots \to B_t$, where B_{n+1} is obtained by swapping two adjacent variables in B_n, and we let $\Pi_s \equiv \Pi_0 \to \Pi_1 \to \cdots \to \Pi_t$, where $B_n \sim \Pi_n$, be the corresponding sequence of orders. Theorem 1 requires

TABLE I
THE 12 CASES FOR THE PROOF OF THEOREM 2.

Order in Π_t	**Swappable in Π_n** $v_i \preccurlyeq v_j \preccurlyeq v_k \preccurlyeq v_l$		**Swappable in Π_{n+1}** $v_i \preccurlyeq v_k \preccurlyeq v_j \preccurlyeq v_l$		Δ
$v_i \prec v_k \prec v_j \prec v_l$	$v_j \preccurlyeq v_k$				-1
$v_i \prec v_k \prec v_l \prec v_j$	$v_j \preccurlyeq v_k$			$v_j \preccurlyeq v_l$	0
$v_i \prec v_l \prec v_k \prec v_j$	$v_j \preccurlyeq v_k$	$v_k \preccurlyeq v_l$		$v_j \preccurlyeq v_l$	-1
$v_k \prec v_i \prec v_j \prec v_l$	$v_j \preccurlyeq v_k$		$v_i \preccurlyeq v_k$		0
$v_k \prec v_i \prec v_l \prec v_j$	$v_j \preccurlyeq v_k$		$v_i \preccurlyeq v_k$	$v_j \preccurlyeq v_l$	+1
$v_k \prec v_j \prec v_i \prec v_l$	$v_i \preccurlyeq v_j$ $v_j \preccurlyeq v_k$		$v_i \preccurlyeq v_k$		-1
$v_k \prec v_j \prec v_l \prec v_i$	$v_i \preccurlyeq v_j$ $v_j \preccurlyeq v_k$		$v_i \preccurlyeq v_k$		-1
$v_k \prec v_l \prec v_i \prec v_j$	$v_j \preccurlyeq v_k$		$v_i \preccurlyeq v_k$	$v_j \preccurlyeq v_l$	+1
$v_k \prec v_l \prec v_j \prec v_i$	$v_i \preccurlyeq v_j$ $v_j \preccurlyeq v_k$		$v_i \preccurlyeq v_k$	$v_j \preccurlyeq v_l$	0
$v_l \prec v_i \prec v_k \prec v_j$	$v_j \preccurlyeq v_k$	$v_k \preccurlyeq v_l$		$v_j \preccurlyeq v_l$	-1
$v_l \prec v_k \prec v_i \prec v_j$	$v_j \preccurlyeq v_k$	$v_k \preccurlyeq v_l$	$v_i \preccurlyeq v_k$	$v_j \preccurlyeq v_l$	0
$v_l \prec v_k \prec v_j \prec v_i$	$v_i \preccurlyeq v_j$ $v_j \preccurlyeq v_k$	$v_k \preccurlyeq v_l$	$v_i \preccurlyeq v_k$	$v_j \preccurlyeq v_l$	-1

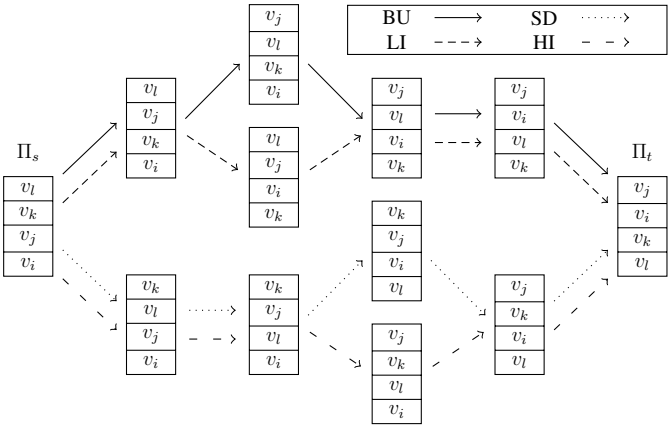

Fig. 2. Sequences of variable orders produced by BU, SD, LI, and HI.

at least $I(\Pi_s, \Pi_t)$ swaps to transform B_s into B_t, and we restrict our study to schedules that use this minimum number of swaps, by always choosing to swap variables forming a swappable inversion. Even with this restriction, identifying an efficient schedule is still non-trivial as there are usually many swappable inversions to choose from at each step.

B. Scheduling heuristics

We begin by proposing five intuitive scheduling heuristics:

- *Random (RAN)*: Randomly choose a swappable inversion.
- *Bring Up (BU)*: Choose the swappable inversion with the highest variable in Π_t not yet in its final position.
- *Sink Down (SD)*: Choose the swappable inversion with the lowest variable in Π_t not yet in its final position.
- *Lowest Inversion (LI)*: Choose the lowest swappable inversion.
- *Highest Inversion (HI)*: Choose the highest swappable inversion.

Fig. 2 illustrates different variable orders produced by these heuristics (except RAN). None of these heuristics considers the cost of a given swap, which is proportional to the number of nodes associated with the top variable being swapped. We then propose a greedy heuristic that considers execution time,

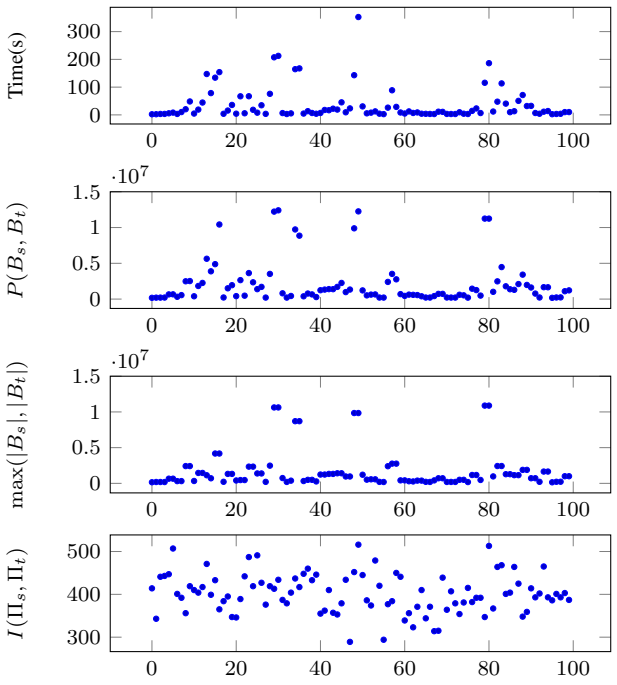

Fig. 3. Results of transforming BDDs of C499 100 times with RAN.

hoping to approach a global optimum by always choosing a local optimum.

- *Lowest Cost (LC)*: Choose the swappable inversion where the variable on top has the fewest associated nodes.

To investigate the metrics affecting the performance of variable reordering, we consider three factors: $\max(|B_s|, |B_t|)$, the largest between the number of nodes before and after reordering; $P(B_s, B_t)$, the peak number of nodes during reordering; and $I(\Pi_s, \Pi_t)$, the required number of swaps. As a preliminary experiment (further experiments are discussed in Section IV), we build the BDD encoding all the outputs of combinatorial circuit C499 [23], generate 100 random variable orders, and transform this BDD into each variable order, sequentially, with the RAN heuristic. Fig. 3 shows the execution time, $P(B_s, B_t)$, $\max(|B_s|, |B_t|)$, and $I(\Pi_s, \Pi_t)$, for each reordering.

The execution time has a high correlation (0.9406) with $P(B_s, B_t)$ and almost as high a correlation (0.868) with $\max(|B_s|, |B_t|)$, but a much lower correlation (0.3476) with $I(\Pi_s, \Pi_t)$. This strengthens the intuition that BDD operations benefit from being applied to BDDs of small size, indicating that it might be useful to seek a schedule that takes $P(B_s, B_t)$ into account. Therefore, we propose a heuristic aimed at minimizing the number of nodes after each swap:

- *Lowest Memory (LM)*: Choose the swappable inversion that will result in the smallest BDD next.

We are not aware of a way to inexpensively and accurately predict the resulting number of nodes after a swap, thus we implement LM by *probing* (actually performing) each swappable inversion, recording the number of resulting nodes, and then undoing the swap. Because of this overhead, LM should

be expected to have poor time performance compared with the other heuristics, but it could provide a good benchmark in terms of achieving a small peak size.

We enhance the performance of LM, by observing that we only need to probe every swappable inversion, and that, if the swappable inversion resulting in the lowest memory happens to be the last one to be considered, it does not need to be reverted, it can be simply kept as the result of that step. More importantly, Theorem 2 implies that, once LM determines which swappable inversion should be performed, the next probing can reuse most of the probes from the previous step, since all the swappable inversions remain the same except the ones that involve the variables just swapped (at most two new ones). In other words, by maintaining appropriate information (i.e., not the absolute size of the BDD, but the difference in its size, which can be positive or negative, computed when probing each swappable inversion), we can identify the swappable inversion to be performed at each step with moderate overhead compared to non-probing heuristics.

To avoid LM's probing overhead, we can *estimate* the number of nodes after a swap. Let N_i be the set of nodes associated with variable v_i, and $\phi(p)$ be the number of incoming edges to node p. Define the *average reference count* of v_i as

$$\overline{\phi}(i) = \sum_{p \in N_i} \phi(p)/|N_i| \tag{3}$$

and, for variables v_i and v_j with $v_i \preccurlyeq_{\Pi_n} v_j$, let $\overline{\phi}(i,j) = \overline{\phi}(i)$. We hypothesize that a low $\overline{\phi}(i,j)$ indicates that swapping v_i and v_j reduces the number of nodes or, at worst, increases it only slightly. $\overline{\phi}$ is a rough estimate because redundant nodes will be reduced and do not contribute to the sum of reference counts. Since $\phi(p)$ is already maintained by BDD libraries that use reference counts to determine when a node is no longer needed, computing $\overline{\phi}$ is inexpensive, and we investigate the effectiveness of the following heuristic:

- *Lowest Average Reference Count (LARC)*: Choose the swappable inversion with the lowest $\overline{\phi}(i,j)$.

We stress that we only consider greedy, no lookahead, heuristics that perform the minimum number $I(\Pi_s, \Pi_t)$ of swaps and differ only in how they schedule these swaps. We ignore heuristics that may perform more swaps, even if it is conceivable that a schedule may be suboptimal in number of swaps (i.e., it may perform swaps that *increase* the number of inversions by one and need to be undone later) yet have better runtime or memory requirements.

IV. Experiments

We implemented our scheduling heuristics as well as the Global Rebuilding algorithm (GR) [3] in a decision diagram library, MEDDLY [2]. We assess memory consumption in terms of BDD nodes, these results are independent of our specific implementation. We set the memory consumption of GR as $|B_s| + |B_t|$, this is a lower bound for rebuilding approaches, as they also require extra storage, e.g. compute table, while swap-based approaches do not. We limit the number of BDD nodes to 1.6×10^8, and impose no time limit.

To compare the set H of heuristics, we normalize the result $x_{r,h}$ (time or memory) of applying heuristic $h \in H$ to a reordering instance r through the following scoring scheme:

$$s_{r,h} = \begin{cases} min_{k \in H}(x_{r,k})/x_{r,h} & \text{if the experiment completes} \\ 0 & \text{otherwise (out of memory)} , \end{cases}$$

i.e., each heuristic earns a score (a value in $(0,1]$) according to its performance relative to the best one, and a score of 0 if it runs out of memory.

The overall score of heuristic h over a set R of reordering instances is: $s_h = \sum_{r \in R} s_{r,h}$, thus a high s_h indicates that heuristic h tends to do well on most reordering instances.

A. Combinatorial circuits

Our first benchmark focuses on combinatorial circuits [23]. For each circuit, we build the BDD encoding all its outputs. Then, we generate 100 random variable orders and transform the BDD from each random variable order to the next, sequentially. The tops of Table II and Table III show the results.

Clearly, for combinatorial circuits, swap-based approaches run significantly faster, while using less memory than GR in most cases. LARC gets high scores for both time and memory — it may not always be the best, but it is consistently close to the best. Fig. 4 compares LARC to the other heuristics using logscale scatter plots for each reordering instance. A data point above the diagonal means that LARC performs better, a data point below means the opposite. In most plots, LARC has a slight advantage over other heuristics.

Unfortunately, in most cases, we do not observe a global optimum of time from LC. On the other hand, with relatively low memory, LM does not pay a heavy price in terms of time, perhaps reflecting the high correlation between runtime and peak memory. In situations where the available memory is limited, LM's probing strategy can be quite valuable.

B. Petri nets

Our second set of experiments uses four Petri net models [13]; we compute the BDD representing each reachable state space and then transform it into different variable orders:

- **phils** models the dining-philosophers problem, where N philosophers sit around a table with a fork between each of them. A philosopher must acquire both the fork to his left and the one to his right before eating.
- **robin** models a round-robin resource sharing protocol, where N processes can cyclically access a resource.
- **slot** models a slotted ring transmission protocol, where, as in **robin**, a token cyclically grants access to the resource. Unlike **robin**, the resource is not modeled, thus each process communicates with its two neighbors through shared transitions, not a globally shared place.
- **queen** models the placement of N queens on an $N \times N$ chessboard such that they are not attacking each other.

Prior to BDD generation, the variable orders are computed using well-known static variable ordering heuristics that exploit known relationships between variables (related variables should generally be close to one another; for Petri nets,

TABLE II
SCORES W.R.T. TIME (LARGER IS BETTER).

Model	#States	BU	SD	LI	HI	LC	LM	LARC	RAN	GR
C432		62.4	67.6	54.0	57.4	71.9	50.9	**72.5**	62.5	3.4
C499		61.7	66.1	61.7	64.2	66.8	52.4	71.9	**73.3**	0.2
C880		70.1	77.3	63.5	**78.6**	72.3	56.7	68.0	70.6	2.4
C1355		55.7	69.4	56.7	76.1	59.4	59.8	**79.6**	56.2	0.2
C1908		79.1	77.0	79.8	83.8	**88.9**	57.7	82.5	76.7	0.9
Sum		329.0	357.4	315.7	360.1	359.3	277.5	**374.5**	339.3	7.1
phils 30	$6.44 \cdot 10^{18}$	58.1	**83.0**	61.0	81.4	(40) 41.1	(34) 24.9	(29) 43.5	(42) 40.5	0.5
phils 40	$1.19 \cdot 10^{25}$	61.5	71.7	60.4	**75.1**	(32) 52.4	(32) 32.7	(32) 51.3	(32) 47.6	0.1
robin 15	$1.10 \cdot 10^{6}$	**85.4**	48.1	85.0	48.7	46.0	40.9	47.0	41.1	0.1
robin 16	$2.35 \cdot 10^{6}$	**80.2**	51.2	79.0	51.7	47.9	50.2	59.8	41.6	0.1
slot 15	$1.46 \cdot 10^{15}$	78.5	61.4	**78.8**	65.0	68.1	60.3	70.4	61.2	0.0
slot 20	$2.73 \cdot 10^{20}$	**65.4**	40.7	63.2	39.6	(19) 38.9	49.4	47.2	(13) 33.1	0.0
queen 7	552	76.9	87.6	76.2	82.5	**88.4**	51.6	84.8	82.7	0.1
queen 8	$2.05 \cdot 10^{3}$	78.6	84.7	74.6	**87.9**	87.7	47.9	82.8	82.8	0.0
Sum		**584.6**	528.4	578.2	531.9	470.5	357.9	486.8	430.6	0.9
phils 70	$7.71 \cdot 10^{43}$	58.2	72.8	53.7	**77.7**	56.0	42.2	75.2	56.4	-
phils 80	$1.43 \cdot 10^{50}$	55.6	72.8	55.6	**74.7**	(18) 56.5	(18) 42.1	(18) 70.3	(18) 62.4	-
phils 90	$2.67 \cdot 10^{56}$	51.3	71.3	52.5	71.4	52.0	48.8	**76.0**	53.6	-
robin 18	$1.06 \cdot 10^{7}$	**66.6**	50.7	65.6	50.5	27.7	40.9	39.4	28.3	-
robin 20	$4.71 \cdot 10^{7}$	**78.0**	50.5	74.9	55.5	52.9	35.6	49.1	(1) 52.3	-
robin 22	$2.07 \cdot 10^{8}$	74.9	75.4	73.3	**79.1**	(6) 45.8	36.4	54.2	(6) 44.9	-
slot 25	$5.26 \cdot 10^{25}$	48.4	50.2	46.7	49.1	(50) 34.2	**55.9**	(4) 39.6	(49) 33.4	-
slot 30	$1.03 \cdot 10^{31}$	**69.3**	56.4	59.9	59.6	(18) 67.1	53.0	(1) 57.8	(18) 59.9	-
slot 35	$2.06 \cdot 10^{36}$	54.8	58.3	51.7	57.1	(32) 45.5	**61.7**	(13) 53.1	(32) 42.9	-
queen 10	$3.55 \cdot 10^{4}$	63.9	75.7	62.9	**82.4**	68.9	40.5	63.4	53.9	-
queen 11	$1.66 \cdot 10^{5}$	62.0	**78.7**	64.1	76.7	72.0	40.7	71.0	71.3	-
queen 12	$8.56 \cdot 10^{5}$	63.9	**86.3**	63.3	84.3	74.9	44.0	79.0	68.4	-
Sum		746.8	798.9	724.3	**817.9**	653.3	541.9	728.1	627.6	-

variables are related if they correspond to places connected via transitions). More precisely, these orders are generated using a modified version of the FORCE heuristic [1], resulting in 10 distinct "good" orders for each model (thus we have 10×9 distinct reordering instances for each model). These orders are realistic starting and ending points for our experiment, and provide insight not found with random orders. Results are shown in the middles and bottoms of Table II (numbers in parentheses, if any, indicate how many of the 90 reorderings failed due to excessive memory requirements) and Table III. The value of the parameter N is listed after the model name. We also list the size of the reachable state space.

This time GR has advantage in keeping the peak memory low, since both the initial and final order are statically chosen to be "good". However, it is still the most time-consuming approach, much slower than the others. This is why we could run GR only for instances with small N.

Among the swap scheduling heuristics, the four simple heuristics BU, SD, LI, and HI exhibit superior and reliable performance, but there is no clear winner. Again, LM proves its effectiveness in requiring a low peak memory, and is even the best w.r.t. both time and memory for **slot 25** and **slot 35**. However, its overall runtime score is the lowest among our swap scheduling heuristics, due to its probing overhead.

An interesting observation not reflected in the scores is that some variable orders generated by FORCE happen to be nearly the reverses of some others. Not surprisingly, changing an order into its reverse is expensive and more likely to have the intermediate BDD size explode. Nevertheless, all four simple heuristics BU, SD, LI, and HI can complete these instances.

C. Additional discussion

Unlike swap-based approaches, the performance of rebuilding is predictable because neither time nor space depend on unknown and unpredictable sizes of intermediate BDDs. However, our experiments demonstrate that the swap-based approaches can be much more time-efficient and in many cases require less memory than rebuilding. To reorder variables of a BDD, we thus suggest trying the swap-based approaches first, while rebuilding should be employed only when the amount of memory required by swaps exceeds the available resources..

SD and HI have a strong inherent connection: it is easy to prove that the schedule produced by SD to reorder B_s with Π_t is the reverse of the one produced by HI to reorder B_t with Π_s. The same connection exists between BU and LI. This is why SD and HI (and BU and LI) have similar time and memory performance.

The two sets of experiments we considered represent two important scenarios. The circuit experiment transforms BDDs

TABLE III
SCORES W.R.T. MEMORY (LARGER IS BETTER).

Model	#States	BU	SD	LI	HI	LC	LM	LARC	RAN	GR
C432		76.4	74.8	75.6	74.7	82.6	**89.7**	84.2	83.3	75.1
C499		72.2	81.9	71.2	81.8	73.3	**95.2**	89.6	87.0	75.2
C880		80.9	83.6	80.8	83.8	82.0	**93.0**	86.3	86.1	70.5
C1355		71.1	80.3	70.8	81.5	71.2	**93.5**	87.7	81.7	77.2
C1908		88.5	91.5	88.8	91.6	90.2	**97.2**	94.3	91.8	65.0
Sum		389.1	412.1	387.2	413.4	399.3	**468.6**	442.1	429.9	363.0
phils 30	$6.44 \cdot 10^{18}$	44.9	46.8	44.9	46.8	41.4	44.0	43.5	43.1	**69.2**
phils 40	$1.19 \cdot 10^{25}$	53.9	50.6	53.9	50.6	52.7	57.2	56.6	55.0	**65.8**
robin 15	$1.10 \cdot 10^{6}$	57.7	37.2	57.7	37.2	39.7	52.6	44.0	40.7	**70.2**
robin 16	$2.35 \cdot 10^{6}$	59.4	40.0	59.3	40.1	35.8	60.3	51.7	39.2	**77.2**
slot 15	$1.46 \cdot 10^{15}$	70.5	59.8	70.6	59.5	61.5	**86.0**	68.5	69.6	57.7
slot 20	$2.73 \cdot 10^{20}$	33.7	26.8	33.7	26.8	23.9	45.4	28.7	25.4	**80.8**
queen 7	552	84.6	87.9	84.6	87.9	85.3	**89.5**	87.9	87.1	47.2
queen 8	$2.05 \cdot 10^{3}$	87.7	88.3	87.7	88.3	87.5	**89.6**	88.5	88.0	45.9
Sum		492.4	437.4	492.4	437.2	427.8	**524.6**	469.4	448.1	514.0
phils 70	$7.71 \cdot 10^{43}$	73.4	82.6	73.4	82.6	77.9	87.2	**87.6**	84.0	-
phils 80	$1.43 \cdot 10^{50}$	65.3	**83.2**	65.3	**83.2**	66.7	71.6	71.6	69.9	-
phils 90	$2.67 \cdot 10^{56}$	68.8	77.9	68.8	77.9	72.9	**87.5**	87.1	81.2	-
robin 18	$1.06 \cdot 10^{7}$	63.0	51.3	**63.1**	51.2	27.1	62.6	45.6	32.5	-
robin 20	$4.71 \cdot 10^{7}$	**80.8**	62.4	80.7	62.5	55.5	65.5	59.9	57.6	-
robin 22	$2.07 \cdot 10^{8}$	**72.5**	69.7	72.2	69.9	43.4	59.3	56.4	46.5	-
slot 25	$5.26 \cdot 10^{25}$	46.5	49.7	46.3	50.3	26.2	**73.9**	39.0	35.4	-
slot 30	$1.03 \cdot 10^{31}$	71.9	70.1	72.2	70.3	67.7	**89.3**	70.2	68.7	-
slot 35	$2.06 \cdot 10^{36}$	51.1	59.9	51.2	60.3	45.5	**82.9**	52.1	53.9	-
queen 10	$3.55 \cdot 10^{4}$	84.0	**88.1**	84.0	**88.1**	82.7	86.9	87.5	83.3	-
queen 11	$1.66 \cdot 10^{5}$	86.4	**88.6**	86.4	**88.6**	86.5	88.2	88.3	86.8	-
queen 12	$8.56 \cdot 10^{5}$	83.9	**88.5**	83.9	**88.5**	83.8	86.9	87.4	83.4	-
Sum		847.6	872.2	847.5	873.5	735.9	**941.8**	832.9	783.3	-

between variable orders of unknown quality, while the Petri net experiment transforms BDDs between high-quality orders. Without considering rebuilding, BU, SD, LI, and HI complete all Petri net instances while other heuristics sometimes cause the size of intermediate BDDs to exceed our imposed bound. Implicitly, these intuitive heuristics are inclined to respect Π_s or Π_t. Specifically, by moving the variables down to their positions in Π_t, SD builds the lower portion of B_t in an early stage of the reordering and does not change it anymore, while HI keeps the lower portion untouched until one inversion in it becomes the highest. This tends to works well when Π_s or Π_t is high-quality (when the initial or the final BDD size is expected to be "small"), while greedy heuristics pursuing local optima are more likely to cause an excessive growth in the number of nodes.

Additionally, LC does not perform as well as hoped, suggesting that, instead of the time complexity of a single swap, future work on reordering should focus on the influence of swaps on the number of nodes. This is consistent with the observation that a heuristic is "good" mostly if it requires a low peak memory. LM proves its effectiveness in requiring a low peak memory, and its runtime could be improved if a probing technique that is less expensive but still accurate in estimating the resulting BDD size is discovered. Finally, randomly-generated schedules are undesirable in practice, since LARC outperforms RAN in most cases.

V. CONCLUSION AND FUTURE WORK

We have presented heuristics to perform variable reordering in BDDs through a schedule of adjacent variable swaps. Experimental results have shown that swap-based approaches tend to be faster than rebuilding, and that reducing the peak memory is critical to the performance of reordering. Also, it is apparent that it can be beneficial to choose different reordering heuristics in different scenarios, and additional knowledge about the initial and target variable orders may be helpful in making such choice.

There are several directions for future work. First, considering the case when the target order Π_t is believed to be high-quality, we should investigate new heuristics that tend to respect Π_t. For example, the heuristics could try to keep together, throughout the reordering process, variables that are close to each other in Π_t. Second, new metrics considering both duplicate nodes and redundant nodes can be designed to more accurately estimate the change in the number of nodes due to a swap.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation under grants ACI-1642397 and CCF-0954132.

REFERENCES

[1] Aloul, F.A., Markov, I.L., Sakallah, K.A.: Force: a fast and easy-to-implement variable-ordering heuristic. In: Proceedings of the 13th ACM Great Lakes symposium on VLSI. pp. 116–119. ACM (2003)
[2] Babar, J., Miner, A.S.: Meddly: Multi-terminal and edge-valued decision diagram library. In: 2010 Seventh International Conference on the Quantitative Evaluation of Systems. pp. 195–196. IEEE (2010)
[3] Bern, J., Meinel, C., Slobodová, A.: Global rebuilding of OBDDs avoiding memory requirement maxima. In: Computer Aided Verification. pp. 4–15. Springer (1995)
[4] Bollig, B., Löbbing, M., Wegener, I.: Simulated annealing to improve variable orderings for OBDDs. In: In Int'l Workshop on Logic Synth. Citeseer (1995)
[5] Bollig, B., Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Transactions on Computers 45(9), 993–1002 (1996)
[6] Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on 100(8), 677–691 (1986)
[7] Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys 24(3), 293–318 (Sep 1992)
[8] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2), 142–170 (Jun 1992)
[9] Burch, J., Clarke, E., Long, D., McMillan, K., Dill, D.: Symbolic model checking for sequential circuit verification. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems 13(4), 401–424 (Apr 1994)
[10] Cabodi, G., Quer, S., Meinel, C., Sack, H., Slobodová, A., Stangier, C.: Binary decision diagrams and the multiple variable order problem. In: Proceedings of the 1998 IEEE/ACM international workshop on Logic synthesis. pp. 346–352 (1998)
[11] Ciardo, G., Lüttgen, G., Miner, A.S.: Exploiting interleaving semantics in symbolic state–space generation. Formal Methods in System Design 31(1), 63–100 (Aug 2007)
[12] Ciardo, G., Lüttgen, G., Yu, A.J.: Improving static variable orders via invariants. In: Petri Nets and Other Models of Concurrency–ICATPN 2007, pp. 83–103. Springer (2007)
[13] Ciardo, G., Zhao, Y., Jin, X.: Ten years of saturation: a petri net perspective. In: Transactions on Petri Nets and Other Models of Concurrency V, pp. 51–95. Springer (2012)
[14] Drechsler, R., Becker, B.: Binary decision diagrams: theory and implementation. Springer (1998)
[15] Drechsler, R., Günther, W., Somenzi, F.: Using lower bounds during dynamic bdd minimization. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 20(1), 51–57 (2001)
[16] Ebendt, R., Drechsler, R.: Lower bounds for dynamic BDD reordering. In: Proceedings of the 2005 Asia and South Pacific Design Automation Conference. pp. 579–582. ACM (2005)
[17] Kendall, M.: Rank Correlation Methods. Charles Griffin and Company Limited (1948)
[18] Malik, S., Wang, A.R., Brayton, R.K., Sangiovanni-Vincentelli, A.: Logic verification using binary decision diagrams in a logic synthesis environment. In: Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on. pp. 6–9. IEEE (1988)
[19] Panda, S., Somenzi, F.: Who are the variables in your neighbourhood. In: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design. pp. 74–77. IEEE (1995)
[20] Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design. pp. 42–47 (1993)
[21] Savickỳ, P., Wegener, I.: Efficient algorithms for the transformation between different types of binary decision diagrams. Acta Informatica 34(4), 245–256 (1997)
[22] Tani, S., Imai, H.: A reordering operation for an ordered binary decision diagram and an extended framework for combinatorics of graphs. In: Algorithms and Computation, pp. 575–583. Springer (1994)
[23] Yang, S.: Logic synthesis and optimization benchmarks user guide: version 3.0. Microelectronics Center of North Carolina (MCNC) (1991)

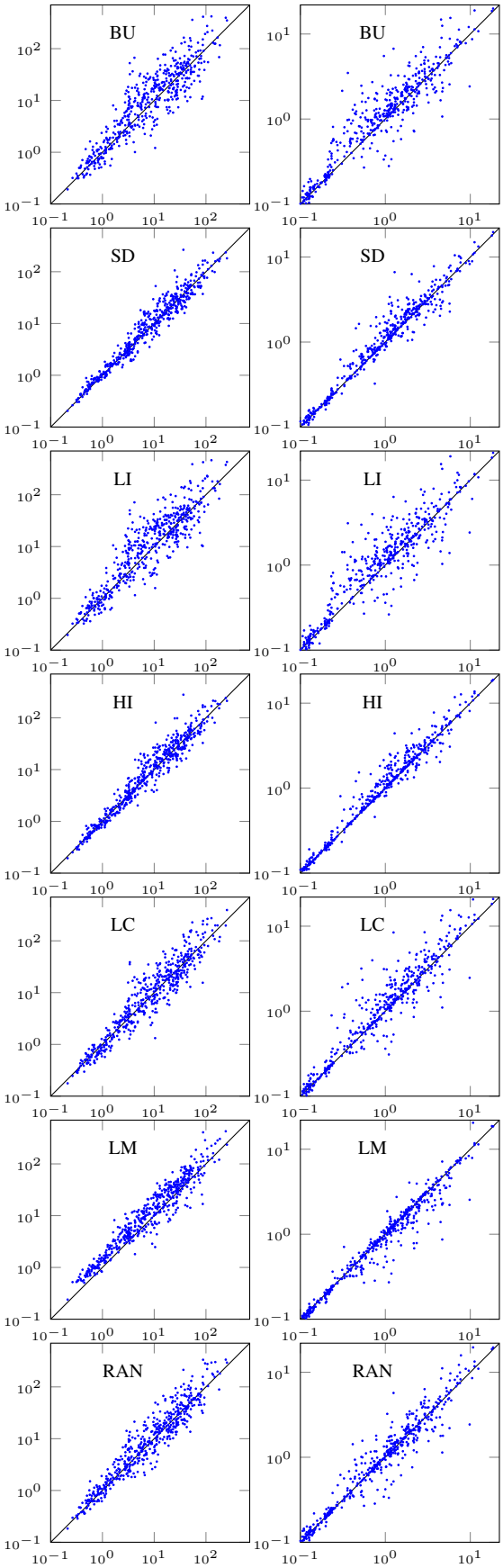

Fig. 4. LARC (x-axis) vs. other heuristics (y-axis) on combinatorial circuits; left: runtime (seconds); right: memory consumption (millions of nodes).