BDDs:
Implementation Issues
&
Variable Ordering

Richard Rudell

Synopsys, Inc.

OBDD Package Interface

typedef struct bdd_manager_struct *bdd_manager;
typedef struct bdd_formula_struct *bdd_formula;

bdd_manager bdd_start();
void bdd_end(bdd_manager bdd);

bdd_formula bdd_create_variable(bdd_manager bdd);
bdd_formula bdd_zero(bdd);
bdd_formula bdd_one(bdd);
bdd_formula bdd_assign(bdd_formula f);
bdd_formula bdd_not(bdd_formula f);
bdd_formula bdd_and(bdd_formula f, bdd_formula g);
bdd_formula bdd_or(bdd_formula f, bdd_formula g);
bdd_formula bdd_and_not(bdd_formula f, bdd_formula g);
bdd_formula bdd_or_not(bdd_formula f, bdd_formula g);
bdd_formula bdd_lte(bdd_formula f, bdd_formula g, bdd_formula h);
bdd_formula bdd_exists(bdd_formula f, bdd_formula g);
bdd_formula bdd_forall(bdd_formula f, bdd_formula g);
bdd_formula bdd_compose(bdd_formula f, bdd_formula g, bdd_formula h);
void bdd_free(bdd_formula f);

int bdd_eq(bdd_formula f, bdd_formula g);
int bdd_le(bdd_formula f, bdd_formula g);
int bdd_disjoint(bdd_formula f, bdd_formula g);
int bdd_ite_test(bdd_formula f, bdd_formula g, bdd_formula h);
int bdd_ex(bdd_formula f, bdd_formula g, bdd_formula h);
OBDD Package Example

cds_manager bdd;
bdd_formula a, b, c, f, g, h, anot, one;

bdd = bdd_start();
one = bdd_one(bdd);

a = bdd_create_variable(bdd);
b = bdd_create_variable(bdd);
anot = bdd_not(a);
f = bdd_and(anot, b);
bdd_free(anot);
c = bdd_create_variable(bdd);
g = bdd_and(a, c);
h = bdd_xor(f, g);
if (bdd_eq(h, one)) {
    "do something"
}
bdd_free(f);
bdd_free(g);
bdd_free(h);
bdd_end(bdd);

---

Review: Chained Hash Table

Insert the pair (key, value) into the hash table

void hash_insert_hash_table *hash_table, void *key, void *value);

Return the value associated with the given key (if it exists)

int hash_lookup_hash_table *hash_table, void *key, void **value);

- density = # entries / # bins
- resize array to maintain constant density (e.g., 4 entries per bin)
- constant-time lookup operation (assuming good hash function)
Review: Memory Function

- Store table of values \( f(x, F(x)) \) for a pure function \( F \).
- Before computing \( F(x) \), check table for stored value
  - avoid re-computing \( F(x) \) if value is already known
  - when \( F(x) \) is computed, save \( (x, F(x)) \) in the table

<table>
<thead>
<tr>
<th>no memory function, exponential complexity</th>
<th>with memory function, linear complexity</th>
</tr>
</thead>
</table>
| int fib(int n) {                          | int fib(int n) {
    int t;                                  |     static int memory[100];
    if (n <= 2) {                            |     int t;
        t = 1;                               |     if (n <= 2) {
    } else {                                 |         t = 1;
        t = fib(n-1) + fib(n-2);             |     } else if (memory[n] != 0) {
    } return t;                             |         t = memory[n];
                                                 |     } else {
}                                                 |         t = fib(n-1) + fib(n-2);
|                                           |     memory[n] = t;
|                                           |     return t;

Multi-Rooted (Shared) OBDD

- A DAG node \( F \) is represented by a tuple \( (x, G, H) \)
  - \( x \) is called the top variable of \( F \)
  - node \( (x, G, H) \) represents the function \( \text{int}(x, G, H) = x, G + \overline{x}, H \)
- DAG contains both external functions (user functions) and internal functions
Unique Table

Hash Table Mapping

\[(x_1, T_i, 1) \rightarrow U_1\]

\[(x_2, T_i, 1) \rightarrow T_1\]

\[(x_2, 0, T_i) \rightarrow T_2\]

\[(x_2, 0, 1) \rightarrow T_3\]

\[(x_{n-1}, \ast, \ast) \rightarrow 1\]

\[(x_{n-2}, \ast) \rightarrow 0\]

- Unique table: hash table mapping tuples \((x, G, H)\) into a node in the DAG
  - before adding a node to the DAG, check to see if it already exists
  - avoids ever creating two nodes with the same function
  - strong canonical form: pointer equality determines function equality
  - resize unique table array to maintain constant density of 4 entries/bin

Shannon Cofactors of an OBDD Function

Computing the Shannon Cofactor (Restriction) on an OBDD function is trivial when the variable is at or above the top variable of the node.

Let \(F = (x, G, H)\) and let \(x_i\) be a variable at level \(i\) or above (i.e., \(j \leq i\)). Then,

\[F_i = \begin{cases} F & \text{if } j < i \\ G & \text{if } j = i \end{cases}\]

Then,

\[F_i = \begin{cases} F & \text{if } j < i \\ H & \text{if } j = i \end{cases}\]

- \(x_2\) is top variable of \(T_1\)
  \[T_{1_x} = T_{1|_{x_1x_2}} = T_3\]

- \(x_2\) is top variable of \(T_2\)
  \[T_{2_x} = T_{2|_{x_1x_2}} = 1\]

- \(x_1\) is above top variable of \(T_1\)
  \[T_{1_x} = T_{1|_{x_1x_2}} = T_1\]

- \(x_1\) is above top variable of \(T_2\)
  \[T_{2_x} = T_{2|_{x_1x_2}} = T_2\]

- \(x_1\) is above top variable of \(T_3\)
  \[T_{3_x} = T_{3|_{x_1x_2}} = T_3\]

- \(x_1\) is above top variable of \(T_4\)
  \[T_{4_x} = T_{4|_{x_1x_2}} = T_4\]

- \(x_1\) is above top variable of \(T_5\)
  \[T_{5_x} = T_{5|_{x_1x_2}} = T_5\]
ITE Recursive Formulation

Let $Z = \text{ite}(F.G.H) = FG + \overline{F}H$. Let $x$ be the top variable of $F,G,H$.

\[
Z = xZ_x + x\overline{x}Z_{\overline{x}} \\
= x(FG + \overline{F}H) + x(FG + \overline{F}H) \\
= x(F,G_x + \overline{F}_xH_x) + x(F,G_x + \overline{F}_xH_x) \\
= \text{ite}(x, \text{ite}(F,G_x,H_x), \text{ite}(F,G_x,H_x)) \\
= (x, \text{ite}(F,G_x,H_x), \text{ite}(F,G_x,H_x)) \]

$\therefore \text{ite}(F,G,H) = (x, \text{ite}(F,G_x,H_x), \text{ite}(F,G_x,H_x))$.

Because $x$ is the top variable of $F,G,H$, the cofactors $F_x, F_{\overline{x}},$ etc. are trivial.

Terminal cases:
- $\text{ite}(1,G,H) = G$
- $\text{ite}(0,G,H) = H$
- $\text{ite}(F,1,0) = F$

Computed Table

- **Computed table**: hash table to implement a memory function for ITE
  - Maps ITE arguments $(F,G,H)$ into the result $\text{ite}(F,G,H)$

- **Computed table is persistent**
  - Computed table results remain valid across top-level calls to ITE
  - allows results computed from previous ITEs to improve performance of subsequent ITEs
  - no need to initialize and free the computed table every ITE
  - initialze computed table once when the OBDD is created
  - saves linear time cost of allocating and freeing the table every ITE
ITE Algorithm

\[
\text{ite}(F, G, H) \{
\text{if (terminal case)} \{ \text{R} = \text{trivial answer}; \}
\text{else if (hash lookup(computed_table, (F,G,H), &result))} \{ \text{R} = \text{result}; \}
\text{else (} \text{x = top variable from F, G, H:} \}
\text{(F1,F0) = trivial_cofactor(F, x);} \text{ } \text{(G1,G0) = trivial_cofactor(G, x);} \text{ } \text{(H1,H0) = trivial_cofactor(H, x);} \text{ } \text{R1 = ite(F1,G1,H1);} \text{ } \text{R0 = ite(F0,G0,H0);} \text{ } \text{if (R1 == R0) } \{ \text{R = R1;} \}
\text{else if (hash lookup(unique_table, (x,R1,R0), &result))} \{ \text{R = result; } \}
\text{else (} \text{R = new node(x,R1,R0);} \text{ } \text{hash_insert(unique_table, (x,R1,R0), R);} \text{ } \text{hash_insert(computed_table, (F,G,H), R);} \}
\text{return R;}
\]

ITE Algorithm Trace:

\[
\begin{align*}
F &= x_1 + x_2 \\
G &= x_1 x_3 \\
H &= x_4 + x_2 \\
\end{align*}
\]

\[
\begin{align*}
I &= \text{ite}(F, G, H) \\
&= (x_1, \text{ite}(F_1, G_1, H_1), \text{ite}(F_2, G_2, H_2)) \\
&= (x_1, \text{ite}(1, C, H), \text{ite}(B_0, 0, H)) \\
&= (x_1, C, (x_1, \text{ite}(B_1, 0, 0, H), \text{ite}(B_0, 0, 0, H))) \\
&= (x_1, C, (x_0, \text{ite}(1, 0, 0, D), \text{ite}(0, 1, 0, D))) \\
&= (x_1, C, (x_0, 0, D))
\end{align*}
\]
ITE Algorithm Improvements

- Improve computed table performance - equivalent forms
  \[ \text{ite}(F, G, 0) = \text{ite}(G, F, 0) = \text{ite}(F, G, F) = \text{ite}(G, F, G) = FG \]
  \[ \text{ite}(F, 1, H) = \text{ite}(H, 1, F) = \text{ite}(F, F, H) = \text{ite}(H, H, F) = F+H \]

- store only 1 of 4 equivalent forms in the computed table
  map to a canonical form (e.g., \text{ite}(F, G, 0) with \text{addr}(F) < \text{addr}(G))

- easy to detect because of strong canonical form

```c
if (F == H || H == 0) {
    /* function is FG */
    H = 0;
    if (address(F) > address(G)) {
        swap(F, G);
    }
}
if (F == U || U == 0) {
    /* function is F + H */
    G = 1;
    if (address(F) > address(H)) {
        swap(F, H);
    }
}
```

Computed Table Cache

- Replace computed table hash table with a hash-based cache
  - store only one entry per bin (no collision chain)
    overwrite existing entry at insert
    check against only one entry at lookup
  - introduces possibility of cache miss which forces redundant computation (affects performance, but not correctness)
    manage impact by sizing the cache proportional to the number of nodes in the unique table

(F, G, H)

![Diagram of computed table cache](image)
Reusing Memory

• New nodes are added to the DAG during ITE
  – minimum number of nodes to represent the result are created!

• The user discards old computation results using bdd_free()
  – problems with deleting the nodes immediately
    1. need to know if the nodes are shared by other roots
    2. computed table entries are never deleted
    3. computed table entries may point at the node
       back-pointers would take too much memory
       sweeping entire computed table would be too slow

---

Garbage Collection

• Solution - Garbage Collection
  – maintain reference count for each node
    includes user references and internal references
    does not count references from the computed table
    reference count is incremented when nodes are reused in the DAG
    reference count is decremented when a root is freed by the user
  – nodes with reference count of 0 are called dead
    they remain in the DAG until the next garbage collection
  – periodic garbage collection
    delete all computed table entries which point to a dead node
    remove all dead nodes from the unique table
Reference Counting Example

- Freeing formula \( U_1 \) reduces reference count on nodes below \( U_2 \)
  - reduce count of \( U_1 \) to 0; it becomes dead so free its children
  - reduce count of \( T_1 \) to 1
  - reduce count of \( T_2 \) to 0; it becomes dead so free its children
  - reduce count of \( T_3 \) to 1
- Nodes \( U_2 \) and \( T_2 \) have ref count 0
  - they will be made available for re-use at the next garbage collection

Effect of Variable Ordering

\[ a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3 \]

Good Ordering

Bad Ordering

Linear Growth

Exponential Growth
OBDD Variable Ordering

- Goal: Form OBDD functions for all nets of a combinational circuit
  - represent function of every net in terms of primary inputs called the global functions
  - first step of verification and optimization algorithms

- OBDDs for all nets or just primary outputs?
  - comb. and seq. verification require only primary output OBDDs
  - optimization algorithms require OBDDs for all nets

- Consistent variable order for all nets?
  - comb. verification can handle different order for each primary output
  - seq. verification and optz algorithms need same order for all nets

- Why worry about variable ordering?
  - using a random variable order almost always fails
    - e.g., OBDDs cannot be formed for 23 of the 35 largest circuits from IWLS'91 benchmark set when using a randomly generated order and 100,000 node limit

OBDDs for Combinational Circuits

- Depth-first walk on combinational circuit from each primary output
  - form logic function for net in terms of primary inputs only

\[
\begin{align*}
T_1 &= \overline{x_1} \\
T_2 &= T_1 \cdot T_2 = \overline{x_1} \cdot x_2 \\
T_3 &= x_1 \cdot \overline{T_2} = 0 \\
T_4 &= \overline{x_2} \\
T_5 &= T_4 \cdot x_3 = x_1 \cdot \overline{x_2} \cdot x_3 + \overline{x_2} \cdot x_3 \\
T_6 &= T_5 \oplus \overline{T_2} = x_1 \cdot \overline{x_2} \cdot x_3 + \overline{x_2} \cdot x_3 \\
z_1 &= T_1 = x_1 \cdot \overline{x_2} \cdot x_3 + \overline{x_2} \cdot x_3 \\
z_2 &= T_2 = \overline{x_1} \cdot x_2 \\
z_3 &= T_3 = x_1 \cdot \overline{T_2} = 0 \\
z_4 &= T_4 = \overline{x_2} \\
z_5 &= T_5 = x_1 \cdot \overline{x_2} \cdot x_3 + \overline{x_2} \cdot x_3 \\
z_6 &= T_6 = T_5 \oplus \overline{T_2} = x_1 \cdot \overline{x_2} \cdot x_3 + \overline{x_2} \cdot x_3
\end{align*}
\]
Heuristic Variable Order

- Use circuit topology to find a good variable order
  - [Fujita-ICCAD88], [Malik-ICCAD88], [Minato-DAC90]
  - variations on the following idea

- Define depth of each node $n$:
  
  $$d(n) = \begin{cases} 
  \max_{f \in \text{Fin}} d(f) + 1 & \text{if } n \text{ not a primary input} \\
  0 & \text{if } n \text{ is a primary input}
  \end{cases}$$

- Starting from deepest output, traverse network in depth-first fashion
  - order fanin at each node by decreasing depth
  - explore deep fanins first
  - break ties arbitrarily (or with more heuristics)

- Order of traversal of primary inputs defines OBDD variable order
  - first variable visited is at the top of the OBDD

---

Heuristic Variable Order Example

- Deepest output is $z_2$
- Depth-first traversal, ordered by depth, visits inputs in order: $x_1, x_2, x_3$
Rationale for Depth-First Heuristic

Primary inputs which feed deep cones of logic get ordered near the top - heuristic: they are the more important decision makers.

Theorem: if $F = G(F_1, F_2, ..., F_n)$ where each pair of functions $F_i$ and $F_j$ share no variables in common, then there exists an optimum OBDD variable order for $F$ which consists of a noninterleaved concatenation of the optimum variable orders of the $F_i$ (for some ordering of the functions $F_1, F_2, ..., F_n$).

Order list = nodes sorted in decreasing levels:

compute level(n); for each node n in list if level heuristic *
Special Case: Fanout Free Circuits

Theorem: If G is AND or OR, then there exists an optimum variable order for a combinational circuit with no reconvergent fanout composed from simple gates (AND, OR, NOT).

Corollary: The depth-first ordering algorithm returns an optimum order for F, which consists of an arbitrary (order-independent) noninterleaved concatenation of the optimum variable orders of the fi.

Optimum orders:
- (a b c d e f g h)
- (h g f e d c b a)

Lemma: If a function f can be written in the form
Heuristic Variable Ordering Limitations

- Random orders almost always fail
  - fails for 23 of 35 largest examples in IWLS'91 benchmark set

- Depth-first heuristic order also fails for many examples
  - fails for 11 of 35 largest examples in IWLS'91 benchmark set

- Is this inherent OBDD exponential complexity or just bad orders?

- Many functions exhibit behavior that some orders produce large OBDDs while other orders produce small OBDDs
  - e.g., n-bit adder
    \[(a_{n-1}, b_{n-1}, a_{n-2}, b_{n-2}, \ldots, a_1, b_1, a_0, b_0)\] linear
    \[(a_{n-1}, a_{n-2}, \ldots, a_1, a_0, b_{n-1}, b_{n-2}, \ldots, b_1, b_0)\] exponential
  - e.g., Achilles Heel function: \[f = x_0.x_1 + x_2.x_3 + \cdots + x_{n-2}.x_{n-1}\]
    \[(x_0, x_1, x_2, x_3, \ldots, x_{n-2}.x_{n-1})\] linear
    \[(x_0, x_2, x_3, \ldots, x_{n-2}.x_1, x_3, \ldots, x_{n-1})\] exponential

Dynamic Variable Ordering

- Motivation:
  - many OBDD operations run out of memory using heuristic ordering
  - programmer must devise application-specific ordering algorithms for each OBDD application
    can be complex for some applications
    expend effort on better heuristics or finding a non-OBDD solution?

- Solution: Dynamic Variable Ordering
  - allow the OBDD package to modify the order on the fly
    OBDD package hides all variable ordering details from user
    OBDD order is no longer static
    Allow OBDD order to change in-between operations
  - maintain consistent order for the OBDD before & after each ITE
    modify the order as a side-effect of OBDD processing
    use current OBDD functions to determine new variable order
Dynamic Variable Ordering Paradigm

- General solution paradigm with many choices
  - when to modify the order?
    e.g., every time the OBDD DAG doubles in size
    e.g., when memory limit is exceeded
    e.g., every 10 ite operations
    e.g., every 100,000 ite steps
  - how to choose a new order
    e.g., variety of OBDD minimization algorithms

- Logically perform variable ordering in-between operations, but:
  - \( F \) and \( G \) are small, but \( F+G \) is too large to be represented
    need to make all 3 functions \( (F,G,F+G) \) small simultaneously
  - solution: reorder variables deep in ITE recursion
    include partial result for \( F+G \)

OBDD Minimization

- Problem definition: Given a multi-rooted OBDD DAG, reorder the variables to minimize the number of nodes in the DAG needed to represent all user functions (simultaneously)

- Complexity:
  - \( n! \) permutations (orders) for \( n \) variables
  - brute-force search: \( O(n!2^n) \)
  - dynamic programming search: \( O(n^23^n) \) [Friedman & Supowit]

- Optimum ordering problem is NP-hard

- Don't need optimum order!
  Just want to avoid exponentially-sized worst case when possible
Adjacent Variable Swap

- Swapping the order of two adjacent variables
  - affects only the nodes at the two levels!

- For a single OBDD function $F$, the nodes at level $i$ represent the unique functions from the set $\{F_{x_1 \cdots x_i}, F_{x_1 \cdots x_{i-1}}, \ldots, F_{x_1 \cdots x_n}\}$ which depend on $x_i$.

For all levels above $x_i$,
the set of cofactors remains unchanged by variable exchange because levels $x_i$ and $x_{i+1}$ are not involved.

For all levels below $x_{i+1}$,
the set of cofactors remains unchanged by variable exchange because of commutivity of cofactor

$\left(F_{x_i}\right)_{x_{i+1}} = \left(F_{x_{i+1}}\right)_{x_i} = F_{x_1 \cdots x_n}$

Adjacent Variable Swap

Before variable swap:

$F = (x_i, G, H) = (x_i, (x_{i+1}, A, B), (x_{i+1}, C, D))$

After variable swap:

$F = (x_{i+1}, G', H') = (x_{i+1}, (x_i, A, C), (x_i, B, D))$
Adjacent Variable Swap Comments

- Several special cases:
  
  \[ G \text{ does not depend on } x_{i+1} \text{ implies } A = B \]
  
  \[ H \text{ does not depend on } x_{i+1} \text{ implies } C = D \]
  
  If \( A = C \), then \( G' = (x_1, A, C) = A \)
  
  If \( B = D \), then \( H' = (x_1, B, D) = B \)

- Modification of \( F \) for the new variable order:
  
  - removes, at most, nodes \( G \) and \( H \) from the DAG  
  
    these nodes may be deleted if they are referenced only by \( F \)
  
  - adds, at most, nodes \( G' \) and \( H' \) to the DAG  
  
    these nodes may be redundant or may already exist in the DAG

Complexity of Adjacent Variable Swap

- Overwrite each node at level \( i \) with a new node at level \( i+1 \)

  \[
  (x_i, G, H) \rightarrow (x_{i+1}, G', H')
  \]
  
  \[
  (x_i, (x_{i+1}, A.B), (x_{i+1}, C.D)) \rightarrow (x_{i+1}, (x_i, A.C), (x_i, B.D))
  \]

- How to reach nodes at level \( i \)?

  - walk DAG from the roots to reach level \( i \) (expensive in run-time)
  
  - double-linked list for all nodes at each level (expensive in memory)
  
  - replace unique table with an array of hash tables, one per level

    replace

    hash_lookup(unique_table, (i, G, H), &value)

    with

    hash_lookup(unique_table[i], (G, H), &value)

    walk down the hash table array for each level to reach all nodes

- Adjacent variable swap complexity is proportional to the number of nodes at level \( i \) and independent of the total DAG size!
Window Permutation Algorithm

- Exhaustive search of all orders within a limited size window

- e.g., variables $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$. Window size 3 starting at $x_1$

  Start $(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$
  swap $(x_3, x_4)$ $(x_1, x_2, x_4, x_3, x_5, x_6, x_7)$
  swap $(x_1, x_3)$ $(x_1, x_2, x_4, x_5, x_3, x_6, x_7)$
  swap $(x_4, x_5)$ $(x_1, x_2, x_5, x_4, x_3, x_6, x_7)$
  swap $(x_4, x_3)$ $(x_1, x_2, x_5, x_3, x_4, x_6, x_7)$
  swap $(x_5, x_3)$ $(x_1, x_2, x_3, x_5, x_4, x_6, x_7)$

- Repeat optimal search within the window at each variable position

[Fujita et al. EDAC'91]
[Ishiura et al. ICCAD'91]
Sifting Algorithm

- Find the optimum position for a variable assuming other variables remain fixed

- e.g., 7 positions for \( x_4 \) (including its current position)

\[
\downarrow x_1 \downarrow x_2 \downarrow x_3 \downarrow x_4 \downarrow x_5 \downarrow x_6 \downarrow x_7
\]

- Use pairwise adjacent swap to exhaustively search all 7 positions

  start \( (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \)

  swap \( x_4, x_5 \) \( (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \)

  swap \( x_4, x_6 \) \( (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \)

  swap \( x_4, x_7 \) \( (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \)

  swap \( x_7, x_4 \) \( (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \)

  swap \( x_6, x_4 \) \( (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \)

  swap \( x_5, x_4 \) \( (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \)

  swap \( x_3, x_4 \) \( (x_1, x_2, x_4, x_3, x_5, x_6, x_7) \)

  swap \( x_2, x_4 \) \( (x_1, x_2, x_3, x_5, x_6, x_7) \)

  swap \( x_1, x_4 \) \( (x_4, x_1, x_2, x_3, x_5, x_6, x_7) \)

Sifting Algorithm Comments

- Sift each variable from its current position
  - down to the bottom of the DAG, then up to the top
  - record best position seen
  - restore best position after completing search

- Advantages:
  - variables can move an arbitrarily long distance
    independent of intermediate increases in the DAG size
  - solves Achilles heel ordering problem optimally starting with bad order
  - solves adder ordering problem optimally starting with bad order
Dynamic Variable Ordering – Results

- Experiment #1: Window Permutation Algorithm vs. Sift Algorithm
  - 35 largest examples from IWLS’91 benchmark set
  - 100,000 node limit placed on the OBDD package
    if memory limit exceeded, example is unsolved
  - Attempt to form OBDD for all primary outputs
    start with heuristic depth-first variable order
    (11 of 35 circuits are unsolved without dynamic ordering)
    apply BDD minimization every time DAG doubles in size

- Results:
  
  window permutation algorithm:
  \[ k=2: \text{solves 2 of 11 unsolved problems} \]
  \[ k=3: \text{solves 3 of 11 unsolved problems} \]
  \[ k=4: \text{solves 3 of 11 unsolved problems} \]
  
  sift algorithm:
  \[ \text{solves 9 of 11 unsolved problems} \]

Dynamic Variable Ordering – Results

- Experiment #2: Heuristic order vs. random order starting point
  - 35 largest examples from IWLS’91 benchmark set
  - 100,000 node limit placed on the OBDD package
    if memory limit exceeded, example is unsolved
  - Attempt to form OBDD for all primary outputs
    start with random variable order
    (23 of 35 examples are unsolved without dynamic ordering)
    apply BDD minimization every time DAG doubles in size

- Compare heuristic ordering start from random order start
  
  sift algorithm:
  \[ \text{solves 32 of 35 examples} \]
  \[ \text{random fails for 1 example which succeeds with heuristic order} \]
  \[ \text{2x longer run-time starting from random order} \]
  \[ \text{slightly larger DAG sizes when starting from random order} \]
Dynamic Variable Ordering Summary

- Effective technique to increase utility and application of OBDDs
  - allows OBDD computation to complete in many cases
  - classic space vs. time trade-off
    no memory increase
    run-time increases up to 10x

- (Almost) removes need for heuristic ordering algorithms

- Sifting algorithm superior to window permutation
  - produces smaller DAG sizes
  - allows more examples to complete

- Dynamic variable ordering future work
  - need to explore other applications to demonstrate utility
  - need faster and more effective BDD minimization algorithms