Automated Reasoning

Lecture 19: Operations on Binary Decision Diagrams (BDDs)

Robert Atkey
bob.atkey@ed.ac.uk

based on originals by Paul Jackson
diagrams from Huth & Ryan, LiCS, 2nd Ed.

Tuesday 24th March 2015

Recap

- Previously:
  - (Reduced, Ordered) Binary Decision Diagrams ((RO)BDDs)
- This time:
  - Operations on ROBDDs
    `reduce`, `apply`, `restrict`, `exists`
  - Symbolic Model Checking with BDDs

Binary Decision Diagrams

Binary Decision Diagrams: DAGs, such that

- Unique root node
- Variables on non-terminal nodes
- Truth-values on terminal nodes
- Exactly two edges from each non-terminal node, labelled 0, 1

Some notation, for a given BDD node n:

- If n is a non-terminal node:
  $\mathrm{var}(n)$ — the variable label on node n;
  $\mathrm{lo}(n)$ — the node reached by following the 0 edge from n;
  $\mathrm{hi}(n)$ — the node reached by following the 1 edge from n;
- If n is a terminal node:
  $\mathrm{val}(n)$ — the truth value labelling n

For a BDD B, the root node is called $\mathrm{root}(B)$.

`reduce`

`reduce` constructs an ROBDD from an OBDD.

1. Label each BDD node n with an integer $\text{id}(n)$,
2. in a single bottom-up pass, such that:
3. two BDD nodes m and n have the same label ($\text{id}(m) = \text{id}(n)$) if and only if m and n represent the same boolean function.

The ROBDD is then created by using one node from each class of nodes with the same label.

reduce

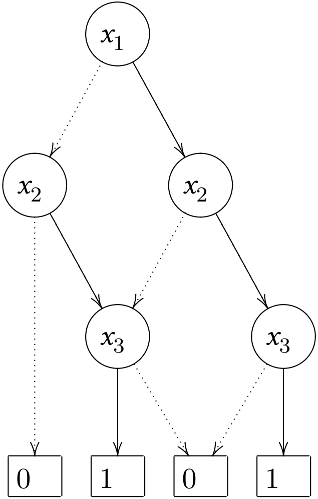
Assignment of labels follows the rules for performing reductions.

To label a node *n*:

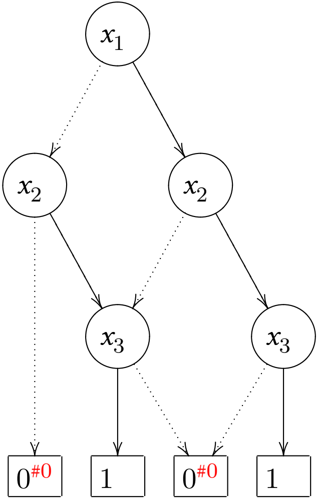
- Remove duplicate terminals:
  if *n* is a terminal node (*i.e.*, $\boxed{0}$ or $\boxed{1}$), then set $\mathrm{id}(n)$ to be $\mathrm{val}(n)$.
- Remove redundant tests:
  if $\mathrm{id}(\mathrm{lo}(n)) = \mathrm{id}(\mathrm{hi}(n))$ then set $\mathrm{id}(n)$ to be $\mathrm{id}(\mathrm{lo}(n))$.
- Remove duplicate nodes:
  if there exists a node *m* that has already been labelled such that
  $$\left\{ \begin{array}{c} \mathrm{var}(m) = \mathrm{var}(n) \\ \mathrm{lo}(m) = \mathrm{lo}(n) \\ \mathrm{hi}(m) = \mathrm{hi}(n) \end{array} \right\}, \text{ set } \mathrm{id}(n) \text{ to } \mathrm{id}(m).$$
  Use a hashtable with $\langle \mathrm{var}(n), \mathrm{lo}(n), \mathrm{hi}(n) \rangle$ keys for $O(1)$ lookup time.
- Otherwise, set $\mathrm{id}(n)$ to an unused number.

Using the "big array" approach to storing BDD nodes, $\mathrm{id}(n)$ is simply the index of the node in the array.

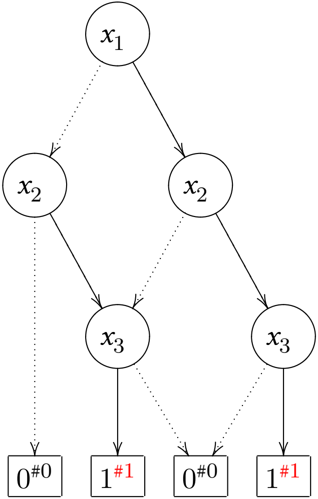
`reduce` Example



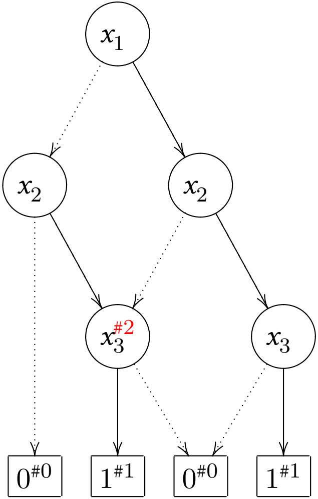
`reduce` Example



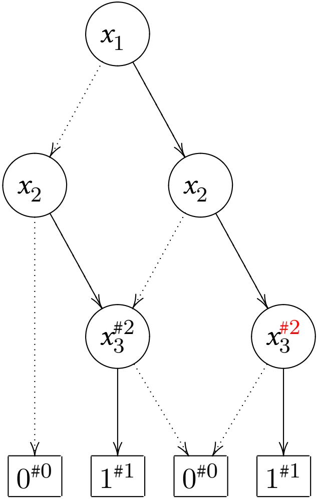
`reduce` Example



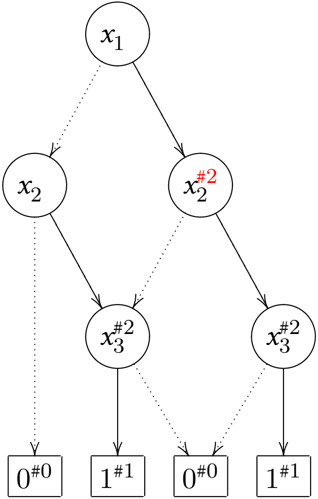
`reduce` Example



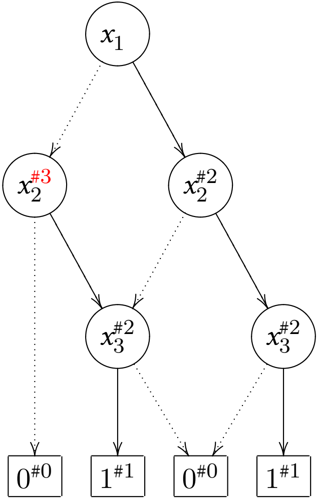
`reduce` Example



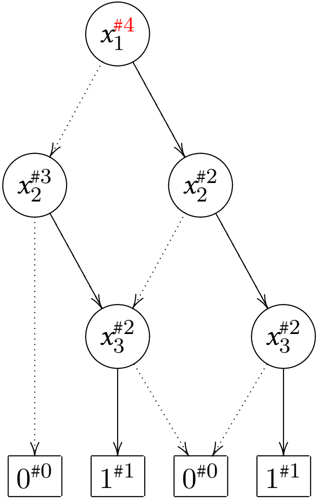
`reduce` Example



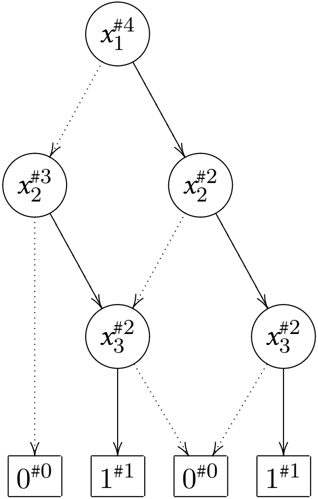
`reduce` Example



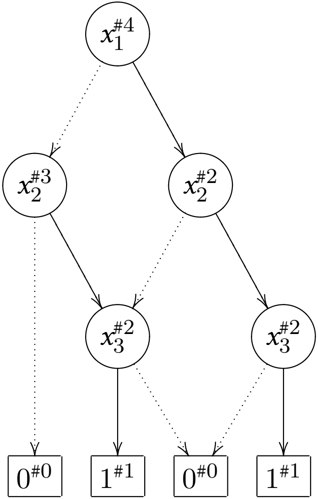
`reduce` Example



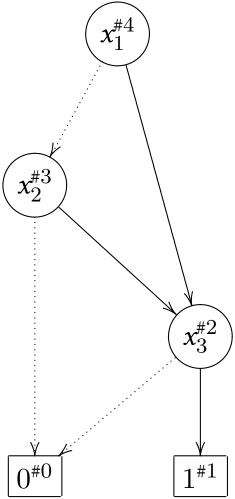
`reduce` Example



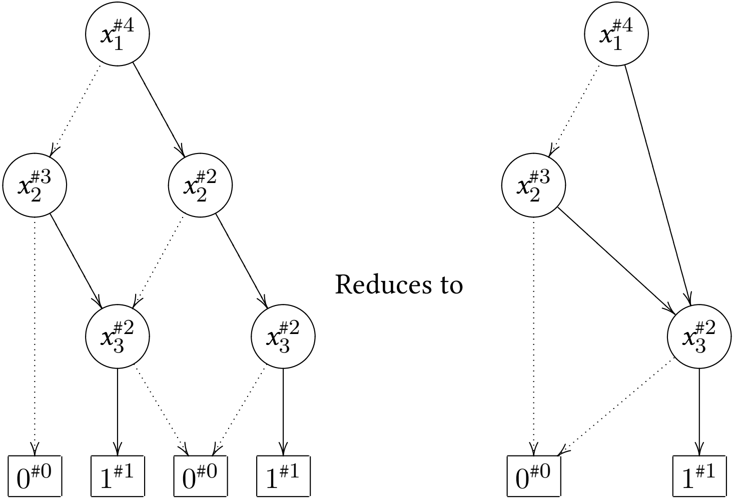
`reduce` Example



Reduces to



`reduce` Example



In practice, labelling and construction are interleaved.

`apply`

Given compatible OBDDs B_f and B_g that represent formulas f and g, `apply`$(\square, B_f, B_g)$ computes a OBDD representing $f \square g$.

- where $\square$ represents some binary operation on boolean formulas
  for example, $\wedge$*,* $\vee$*,* $\oplus$
- Unary operations can be handled too.
  for example, negation: $x \square y = x \oplus 1$

`apply`: Shannon expansions

For any boolean formula f and variable x, it can be written as:

$$f \equiv (\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$$

This is the **Shannon expansion** of f (originally due to G. Boole).

`apply`: Shannon expansions

For any boolean formula f and variable x, it can be written as:

$$f \equiv (\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$$

This is the **Shannon expansion** of f (originally due to G. Boole).

In particular: $f \,\square\, g$ can be expanded like so:

$$f \,\square\, g \equiv (\neg x \wedge (f[0/x] \,\square\, g[0/x])) \vee (x \wedge (f[1/x] \,\square\, g[1/x]))$$

`apply`: Shannon expansions

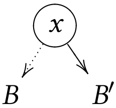
For any boolean formula f and variable x, it can be written as:

$$f \equiv (\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$$

This is the **Shannon expansion** of f (originally due to G. Boole).

In particular: $f \Box g$ can be expanded like so:

$$f \Box g \equiv (\neg x \wedge (f[0/x] \Box g[0/x])) \vee (x \wedge (f[1/x] \Box g[1/x]))$$

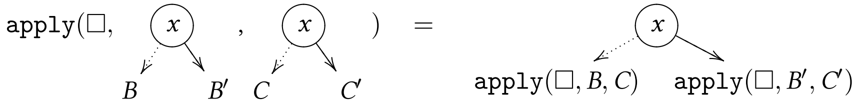
If a BDD 

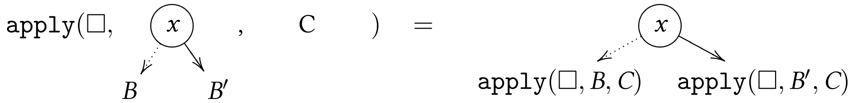
represents a boolean function f, then:

1. B represents $f[0/x]$ and B' represents $f[1/x]$; and
2. The BDD is effectively a compressed representation of f in Shannon normal form.

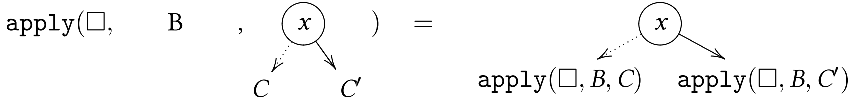
So: implement `apply` recursively on the structure of the BDDs.

apply: cases





when C is terminal node, or non-terminal with $\mathrm{var}(\mathrm{root}(C)) > x$

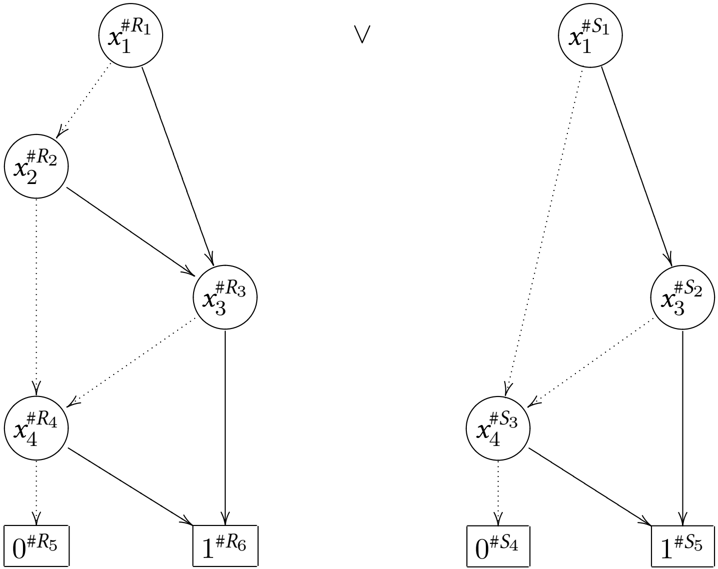


when B is terminal node, or non-terminal with $\mathrm{var}(\mathrm{root}(B)) > x$

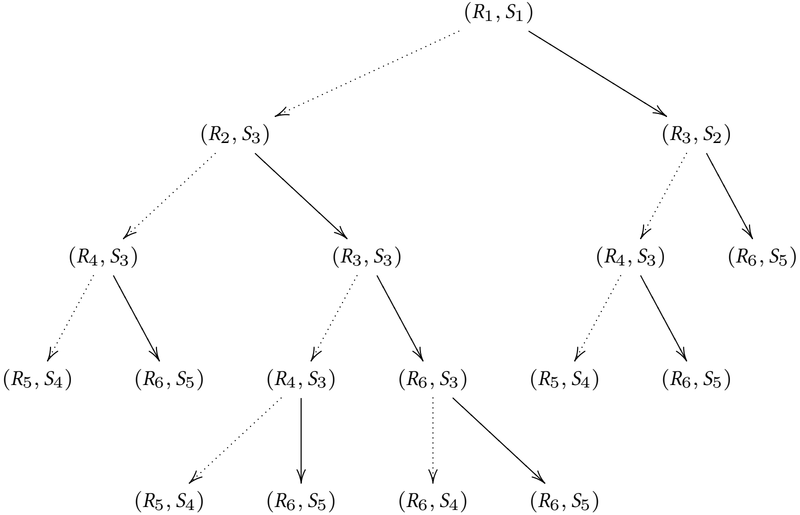
$$\texttt{apply}(\Box, \boxed{u}, \boxed{v}) = \boxed{u \Box v}$$

`apply`: example

Compute `apply`$(\vee, B_f, B_g)$, where B_f and B_g are:



`apply`: recursive calls



`apply`: memoisation

The recursive `apply` implementation will generate an OBBD.

- Apply `reduce` to convert it back to an ROBDD.

However, as can be seen from the tree of recursive calls, there are many calls to `apply` with the same arguments.

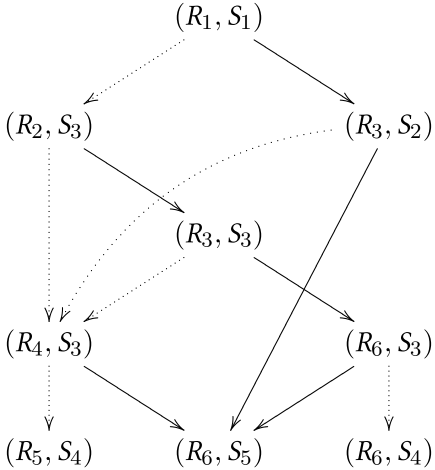
- Each invocation of `apply` where at least one of the arguments is non-terminal generates two further calls to `apply`: the number of calls is worst-case exponential in the sizes of the original diagrams.

We are not taking into account the **sharing** in BDDs.

We can greatly improve the run-time by using **memoisation**: remembering the results of previous calls.

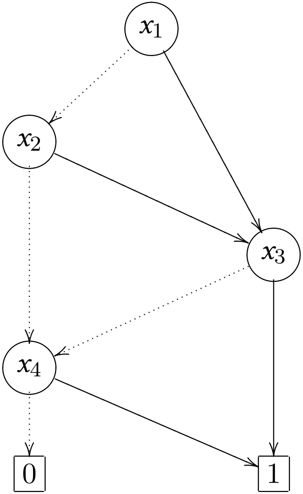
apply: memoised recursive calls

Memoisation results in at most $|B_f| \cdot |B_g|$ calls to `apply`.



`apply`: Result

If we are careful to never create the same BDD node twice (using the same lookup table technique as `reduce`), then with memoisation, we automatically get a reduced BDD:



Other Operations

`restrict`$(0, x, B_f)$ computes ROBDD for $f[0/x]$

1. For each node n labelled with x, incoming edges are redirected to $\mathrm{lo}(n)$, and the node n is removed.
2. Resulting BDD then reduced with `reduce`.
3. (again, `reduce` can be interleaved with the removal.)

`exists`(x, B_f) computes ROBDD for $\exists x.\ f$.

1. Uses the identity

$$(\exists x.\ f) \equiv f[0/x] \vee f[1/x]$$

2. Realised using the `restrict` and `apply` functions:

$$\texttt{apply}(\vee, \texttt{restrict}(0, x, B_f), \texttt{restrict}(1, x, B_f))$$

Time Complexities

Algorithm	Input OBDDs	Output OBDD	Time complexity
`reduce`	B	reduced B	$O(\|B\| \cdot \log \|B\|)$
`apply`	B_f, B_g (reduced)	$B_{f \square g}$ (reduced)	$O(\|B_f\| \cdot \|B_g\|)$
`restrict`	B_f (reduced)	$B_{f[0/x]}$ or $B_{f[1/x]}$ (red'd)	$O(\|B_f\| \cdot \log \|B_f\|)$
$\exists$	B_f (reduced)	$B_{\exists x_1 \ldots x_n . f}$ (reduced)	NP-complete

H&R, Figure 6.23

Implementing CTL Model Checking using BDDs

Recall:

1. CTL model checking computes a set of states $[\![\phi]\!]$ for every sub-formula ϕ of the original formula.
2. Sets of states will be represented using ROBDDs

States are represented by boolean vectors $\langle v_1, \ldots, v_n \rangle$.

Sets of states are represented using ROBDDs on n variables $x_1, \ldots, x_n$ that describe the **characteristic function** of the set.

- Operations on sets are implemented using the operations on BBDs

For example, the definition

$$[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$$

Is implemented by:

$$B_{[\![\phi \wedge \psi]\!]} = \texttt{apply}(\wedge, B_{[\![\phi]\!]}, B_{[\![\psi]\!]})$$

Implementing CTL Model Checking using BDDs

Transition relations $(\to) \subseteq S \times S$ are represented by ROBDDs on $2n$ variables.

- If the variables $x_1, \ldots, x_n$ describe the current state, and the variables $x'_1, \ldots, x_n$ describe the next state, then a good ordering is $x_1, x'_1, x_2, x'_2, \ldots, x_n, x'_n$ (interleaving).

When translating from the model description, the boolean formulas describing the:

1. initial state set
2. transition relation
3. defined variables

are translated into ROBDDs by using the `apply` algorithm, following the structure of the original formula.

This avoids exponential blow-up from first constructing a decision tree and then reducing.

Implementing CTL Model Checking using BDDs

The function applications

$$\begin{array}{rcl} \mathrm{pre}_{\exists}(Y) & \doteq & \{s \in S \mid \exists s' \in S.\, (s \to s') \wedge s' \in Y\} \\ \mathrm{pre}_{\forall}(Y) & \doteq & \{s \in S \mid \forall s' \in S.\, (s \to s') \to s' \in Y\} \end{array}$$

are implemented using BDDs like so:

$$B_{\mathrm{pre}_{\exists}(Y)} = \mathtt{exists}(\overrightarrow{x'}, \mathtt{apply}(\wedge, B_{\to}, B_{Y'}))$$

where

- $B_{\to}$ is the ROBDD representing the transition relation $\to$;
- $B_{Y'}$ is the RODBB representing the set Y with the variables $x_1, \ldots, x_n$ renamed to $x'_1, \ldots, x'_n$.

And:

$$\mathrm{pre}_{\forall}(Y) = S - \mathrm{pre}_{\exists}(S - Y)$$

where $S - Y$ is implemented by negation (via apply).

Implementing CTL Model Checking using BDDs

To implement the temporal connectives, we compute fix points.

$$\begin{array}{rcl} [\![\mathbf{EF}\ \phi]\!] & = & \mu Y.\ [\![\phi]\!] \cup \mathrm{pre}_{\exists}(Y) \\ [\![\mathbf{EG}\ \phi]\!] & = & \nu Y.\ [\![\phi]\!] \cap \mathrm{pre}_{\exists}(Y) \\ \ldots & & \end{array}$$

By Knaster-Tarski, we know that:

- $F^{|S|}(\emptyset)$ is the *least* fixed point of F: $\mu Y.F(Y)$
- $F^{|S|}(S)$ is the *greatest* fixed point of F: $\nu Y.F(Y)$

Compute $[\![\mathbf{EF}\ \phi]\!]$ using the sequence (of ROBDDs)

$$Y^0 = \emptyset,\ Y^1 = [\![\phi]\!] \cup \mathrm{pre}_{\exists}(\emptyset),\ Y^2 = [\![\phi]\!] \cup \mathrm{pre}_{\exists}([\![\phi]\!] \cup \mathrm{pre}_{\exists}(\emptyset)), \ldots$$

Usually, we won't need $|S|$ steps: we can stop when $Y_i = Y_{i+1}$

- This check is very cheap with ROBDDs.

Summary

- Operations on BDDs (H&R 6.2)
  - `reduce`
  - `apply`
  - `restrict`, `exists`
- Symbolic Model Checking (H&R 6.3)
  - Representing states and transitions as BDDs
  - Implementing the CTL MC algorithm with BDDs
- Next:
  - Friday 27th March: Phil Scott
    "Formalising the Foundations of Geometry"
  - Next Tuesday (31st March): Exam Review