BDD operations

Paul Jackson[1]
Paul.Jackson@ed.ac.uk

University of Edinburgh

Formal Verification
Autumn 2023

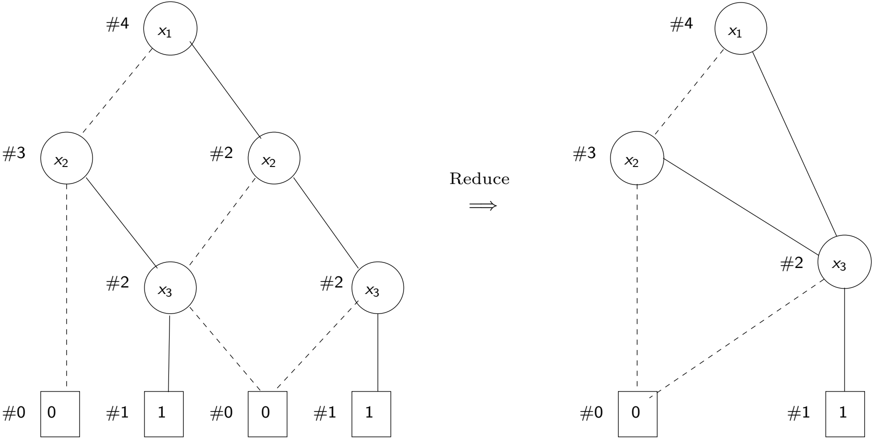
[1] Diagrams from Huth & Ryan, LiCS, 2nd Ed.

`reduce` algorithm

Aim is to construct a ROBDD from an OBDD.

- Adds integer labels $\mathrm{id}(n)$ to each node n of a BDD in a single bottom-up pass
- Key property: if nodes m and n are labelled, then $\mathrm{id}(m) = \mathrm{id}(n)$ iff m and n represent the same Boolean function.
- Rules for adding label to node n:
  - remove duplicate terminals: if n terminal, set $\mathrm{id}(n)$ to $\mathrm{val}(n)$
  - remove redundant tests: if $\mathrm{id}(\mathrm{lo}(n)) = \mathrm{id}(\mathrm{hi}(n))$, set $\mathrm{id}(n)$ to $\mathrm{id}(\mathrm{lo}(n))$
  - remove duplicate nodes: if there exists a labelled node m such that $\left\{\begin{array}{ccc} \mathrm{var}(m) & = & \mathrm{var}(n) \\ \mathrm{id}(\mathrm{lo}(m)) & = & \mathrm{id}(\mathrm{lo}(n)) \\ \mathrm{id}(\mathrm{hi}(m)) & = & \mathrm{id}(\mathrm{hi}(n)) \end{array}\right\}$, set $\mathrm{id}(n)$ to $\mathrm{id}(m)$
    Use hash table with $\langle \mathrm{var}(n), \mathrm{id}(\mathrm{lo}(n)), \mathrm{id}(\mathrm{hi}(n)) \rangle$ keys for $O(1)$ search time
  - otherwise, set $\mathrm{id}(n)$ to unused number
- ROBDD generated by using 1 node from each class of nodes with the same label
- Node sharing between ROBDDs possible if hash table shared

reduce example



`apply` algorithm I - specification

Given

- Boolean formulas f and g,
- ROBDDs B_f and B_g for f and g,
- a binary operation op on boolean formulas (e.g. $\wedge$, $\vee$, $\oplus$)

$$\texttt{apply}(\mathrm{op}, B_f, B_g)$$

computes a ROBDD for f op g.

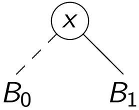
Can also use `apply` for negation: compute the ROBDD for $\neg f$ using

$$\texttt{apply}(\oplus, B_f, \boxed{1})$$

In essence, this just swaps terminal nodes $\boxed{0}$ and $\boxed{1}$.

`apply` algorithm II - the Shannon expansion

Consider a Boolean formula f represented by a BDD with top-level structure



Sub-BDDs B_0 and B_1 also correspond to formulas, say f_0 and f_1

What are the relationships between f, f_0 and f_1?

$$
\begin{aligned}
f_0 &\equiv f[0/x] \\
f_1 &\equiv f[1/x] \\
f &\equiv \overline{x}.f_0 + x.f_1
\end{aligned}
$$

The implied formula

$$f \equiv \overline{x}.f[0/x] + x.f[1/x]$$

is called the *Shannon expansion* of Boolean formula f with respect to the variable x.

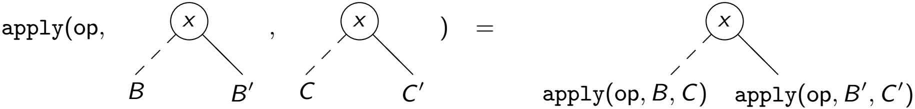
`apply` algorithm III - the key idea

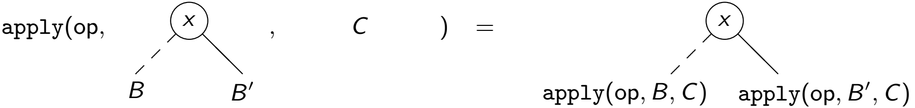
Consider the Shannon expansion of f op g and pushing substitutions through op:

$$
\begin{aligned}
f \text{ op } g &\equiv \overline{x}.(f \text{ op } g)[0/x] + x.(f \text{ op } g)[1/x] \\
&\equiv \overline{x}.(f[0/x] \text{ op } g[0/x]) + x.(f[1/x] \text{ op } g[1/x])
\end{aligned}
$$

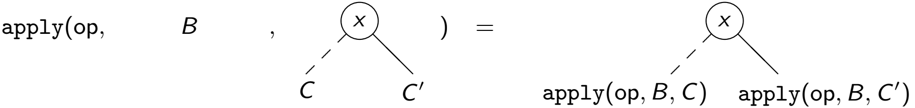
This recursive characterisation of op suggests a recursive algorithm for computing op on BDDs

apply algorithm IV - the definition





where C is 1) a terminal node or 2) a non-terminal with $var(root(C)) > x$

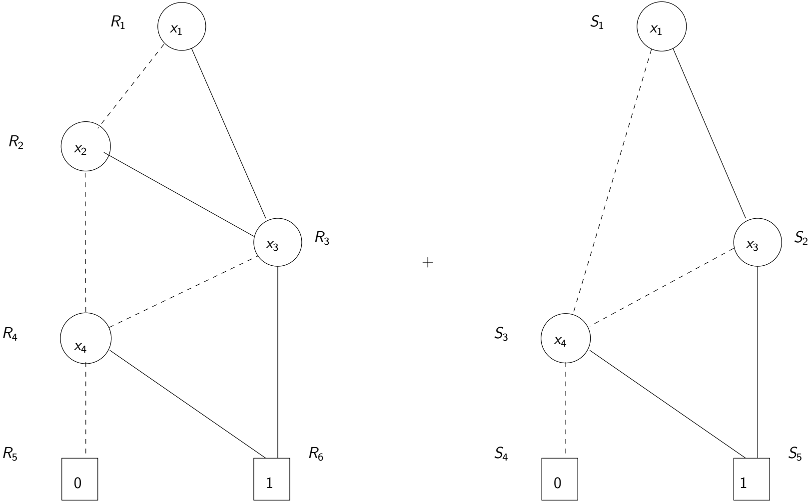


where B is 1) a terminal node or 2) a non-terminal with $var(root(B)) > x$

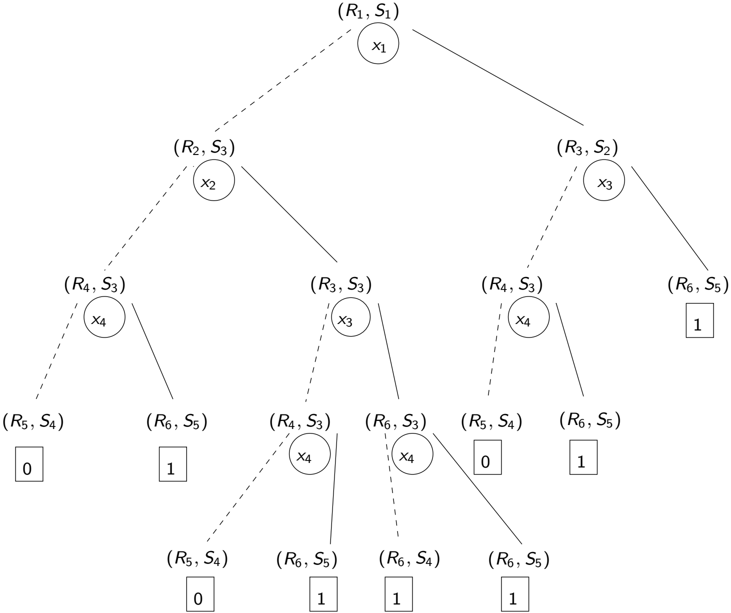
$$\texttt{apply}(\mathrm{op}, \boxed{u}, \boxed{v}) = \boxed{w} \quad \text{where } w = u \text{ op } v$$

`apply` example

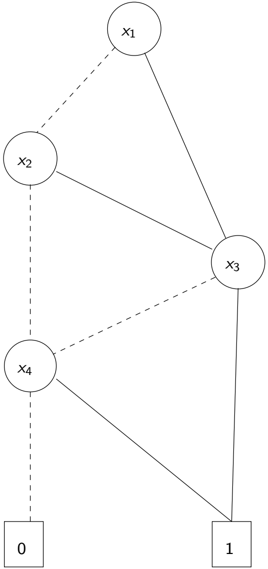
Compute $\texttt{apply}(+, B_f, B_g)$ where B_f and B_g are:



Recursive calls of `apply`



Final result from `apply` execution



`apply` remarks

- In general, result will not be an ROBDD, so need to use `reduce` afterwards
  - Or can incorporate aspects of `reduce` into `apply` so result is always reduced
- Naive implementation has run-time exponential in number of variables.
  - Each apply call in 3 of 4 cases results in two recursive calls
- However, only $|B_f| \cdot |B_g|$ distinct calls
  - If calls *memoized*, $O(|B_f| \cdot |B_g|)$ time complexity is possible.

Other operations

- `restrict`$(0, x, B_f)$ computes ROBDD for $f[0/x]$
  1. For each node n labelled with an x, incoming edges are redirected to $\mathrm{lo}(n)$ and n is removed.
  2. Resulting BDD is reduced.

- `exists`(x, B_f) computes ROBDD for $\exists x.\, f$
  - Uses identity $(\exists x.\, f) \;\equiv\; f[0/x] + f[1/x]$ and `restrict` and `apply` functions

Time complexities

Algorithm	Input OBDD(s)	Output OBDD	Time-complexity
`reduce`	B	reduced B	$O(\lvert B\rvert \cdot \log \lvert B\rvert)$
`apply`	B_f, B_g (reduced)	$B_{f \text{ op } g}$ (reduced)	$O(\lvert B_f\rvert \cdot \lvert B_g\rvert)$
`restrict`	B_f (reduced)	$B_{f[0/x]}$ or $B_{f[1/x]}$ (reduced)	$O(\lvert B_f\rvert \cdot \log \lvert B_f\rvert)$
$\exists$	B_f (reduced)	$B_{\exists x_1.\exists x_2.\ldots.\exists x_n.f}$ (reduced)	NP-complete

H&R Figure 6.23

Encoding CTL algorithms using BDDs I

- States represented using Boolean vectors $\langle v_1, \ldots, v_n \rangle$, where $v_i \in \{0, 1\}$.
- Sets of states represented using BDDs on n variables $x_1, \ldots x_n$ describing characteristic functions of sets.
- Set operations $\cup, \cap, \bar{}$ computed using the `apply` algorithm and the Boolean operations $+, \cdot, \bar{}$.
- Transition relations described using BDDs on $2n$ variables.
  - If Boolean variables $x_1, \ldots x_n$ describe initial state and Boolean variables $x'_1, \ldots x'_n$ describe next state, then good ordering is $x_1, x'_1, x_2, x'_2, \ldots x_n, x'_n$.
- Translations of Boolean formulas describing state sets and transition relations into BDDs make use of `apply` algorithm, following structure of formulas
  - This avoids the intractable exponential blow-up if instead one tried to first construct a binary decision tree.

Encoding CTL algorithms using BDDs II

- The existential pre-image function

$$\mathsf{pre}_\exists(Y) \ \doteq\ \{s \in S \mid \exists s' \in S.\, s \to s' \wedge s' \in Y\}$$

  is computed using

$$\texttt{exists}(x_1', \texttt{exists}(x_2', \ldots \texttt{exists}(x_n', \texttt{apply}(\cdot, B_\to, B_Y')) \ldots))$$

  where
  - $B_\to$ is the ROBDD representing the transition relation $\to$
  - B_Y' is the ROBDD representing set Y with the variables $x_1, \ldots x_n$ renamed to $x_1', \ldots x_n'$
- To compute the universal pre-image function

$$\mathsf{pre}_\forall(Y) \ \doteq\ \{s \in S \mid \forall s' \in S.\, s \to s' \Rightarrow s' \in Y\}$$

  we observe that

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

  and note that the computation for $-$ (*set complement*) is the same as the computation for logical negation.