Algorithms for Reduced OBDDs.

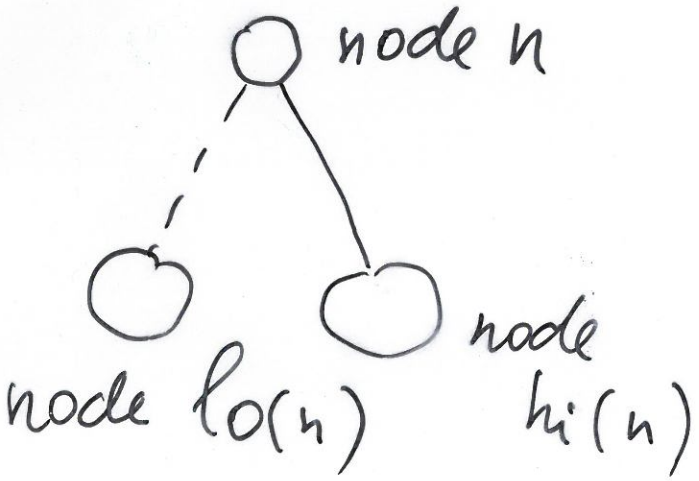
Reduce (Implements C1-C3)

- works bottom-up, layer by layer
  $[X_i, \ldots, X_\ell]$
  $\leq \ell + 1$ layers
- assign equal IDs to nodes representing equiv. bool. functions.

Procedure:

- label terminals: $\boxed{0}$ #0
  $\boxed{1}$ #1.

Notation:



- If $id(lo(n)) = id(hi(n))$ then set $id(n)$ the same.
  (eliminate redundant test, C2)

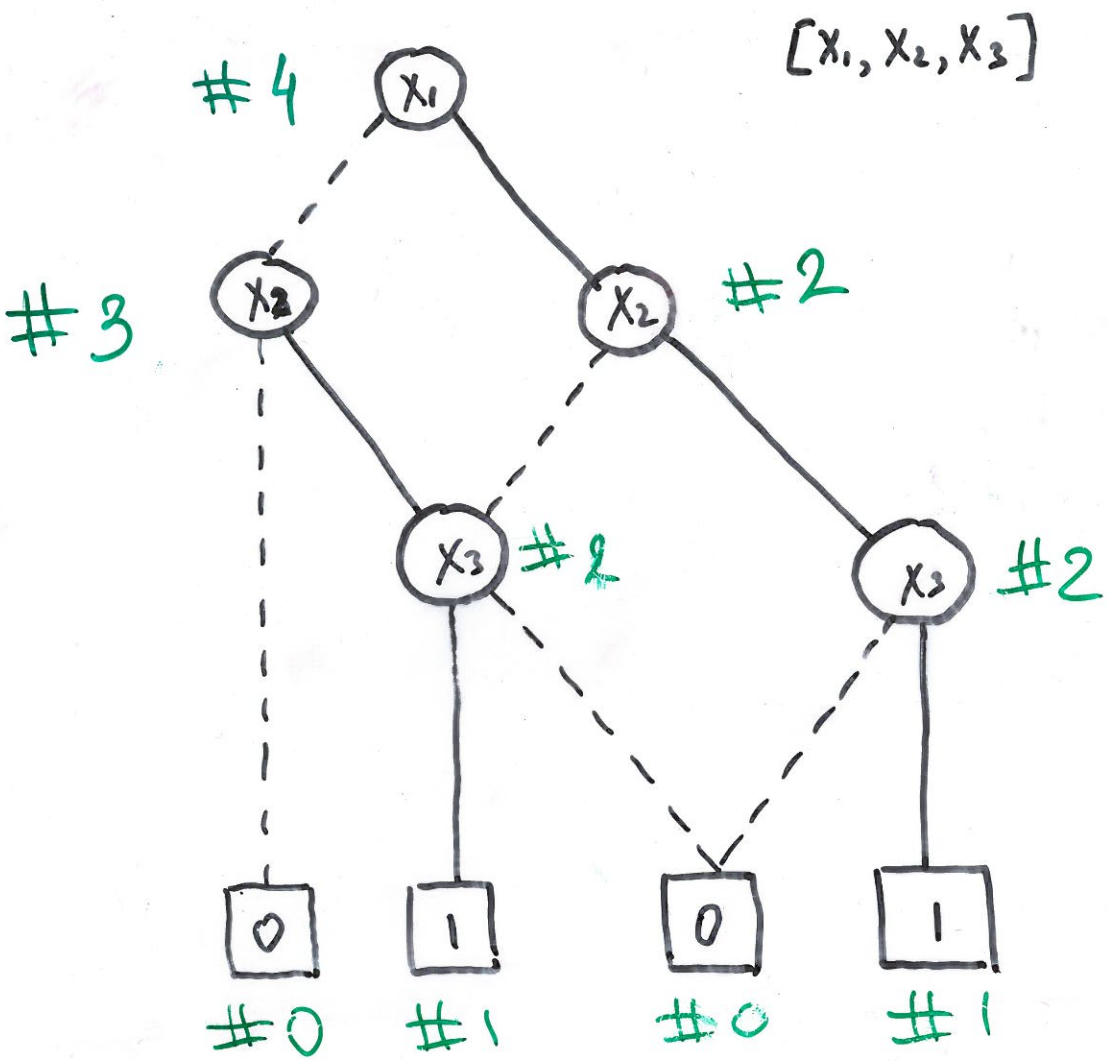
$lo(n)$, $hi(n)$ represent the same bool. func.

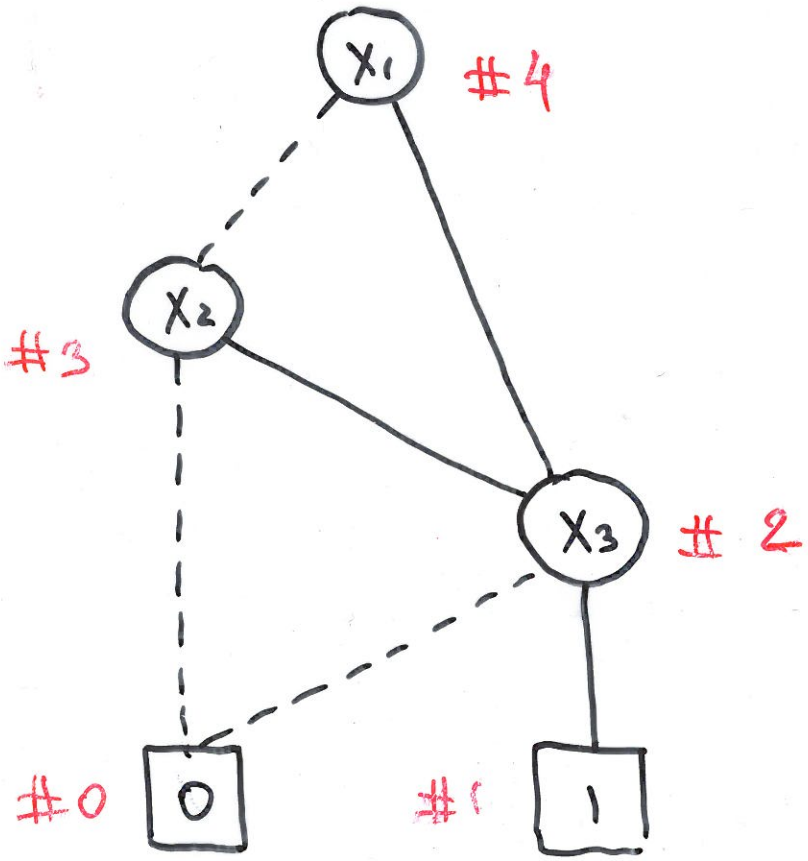
- If $\exists n \; \exists m$ with the same var. x_i, and $id(lo(n)) = id(lo(m))$
  $id(hi(n)) = id(hi(m))$

then set id(n) to be id(m)

(eliminate duplicate nodes, C1 and C3)

- o.w. set $id(n)$ to the next unused integer.


[X1, X2, X3]
#4
X1
#3
X2
X2
#2
X3
#2
X3
#2
0
1
0
1
#0
#1
#0
#1


X1
#4
X2
#3
X3
#2
#0
0
#1
1

Apply

Given B_f, B_g, with a compatible var. ordering.

apply(op, B_f, B_g) generates the reduced OBDD of B_f op B_g,

$op : \{0,1\} \times \{0,1\} \to \{0,1\}$,

any binary bool. function,

e.g. $+, \cdot, \oplus, -$ (via $f \oplus 1$)

High-level Idea: (recursion on the structure of B_f, B_g)

1. Pick var. v, the highest in the ordering, which occurs in both B_f, B_g.
2. Split the problem into 2 subproblems, $v=1$ and $v=0$, solve them recursively.
3. At the leaves, apply op directly.

Notation:

① $f[0/x]$, $f[1/x]$ restrictions of f,
replace each occ. of x in f with 0 (resp. 1).

② $f \equiv g$ f and g represent the same bool. func.

e.g. if $f(x,y) \overset{def}{=} x \cdot (y + \bar{y})$

then $f[0/x](x,y) = 0 \cdot (y + \bar{0}) = 0$

$f[1/y](x,y) = x \cdot (1 + \bar{x}) = x$

Goal: Recursion

Shannon expansion:

$f \equiv \bar{x} \cdot f[0/x] + x \cdot f[1/x]$

f op g

for all f, for all x (not even in f)

"Apply" is based on the Shannon expansion for $f \text{ op } g$:

$$f \text{ op } g \equiv \bar{x}_i \cdot (f[0/x_i] \text{ op } g[0/x_i]) + x_i \cdot (f[1/x_i] \text{ op } g[1/x_i])$$

Let r_f be the root of B_f

r_g — " — B_g

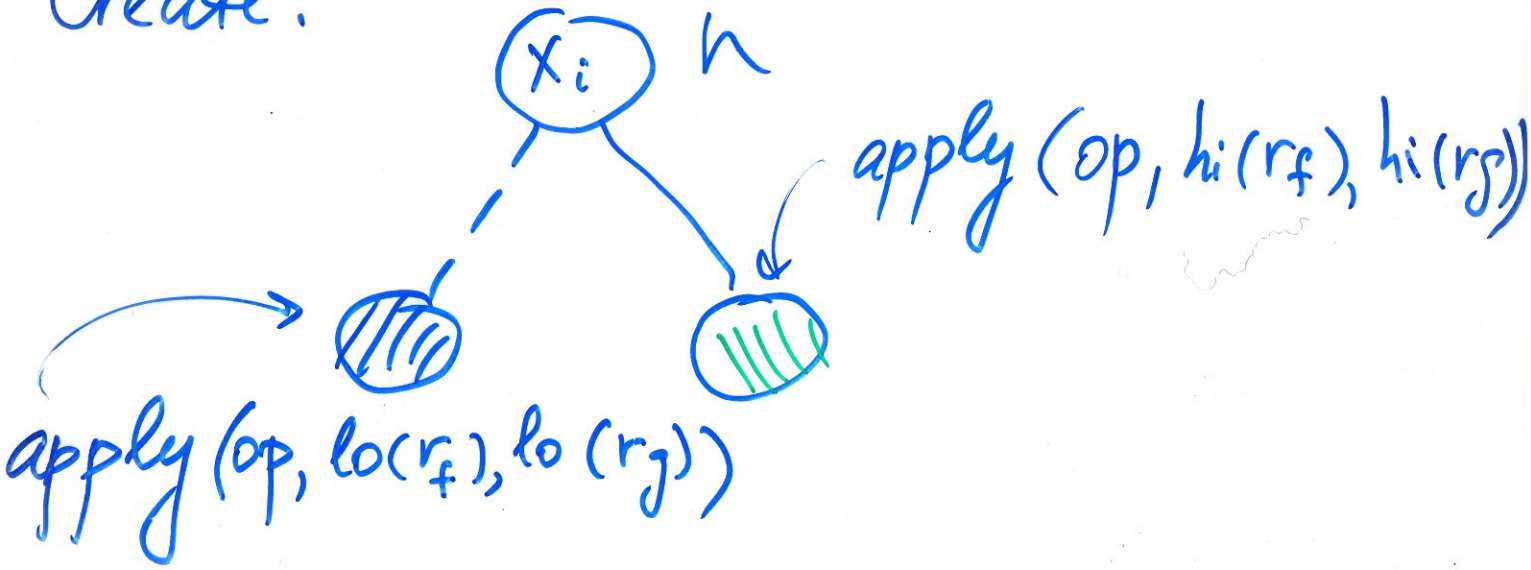
Procedure

1) r_f, r_g are terminals (0,1) with labels l_f, l_g, resp.

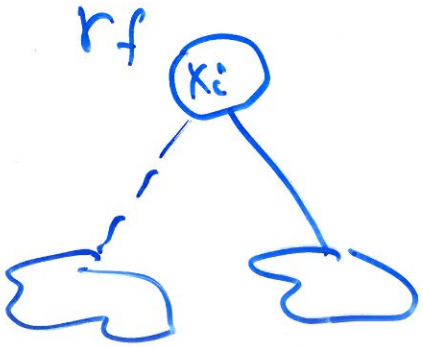
Compute $l_f \text{ op } l_g = \begin{cases} B_0 \text{ if the result is 0} \\ B_1 \text{ if the result is 1.} \end{cases}$

2) r_f, r_g are X_i-nodes

Create:



3) r_f is an X_i-node, r_g is a terminal node or an X_j-node, $j > i$



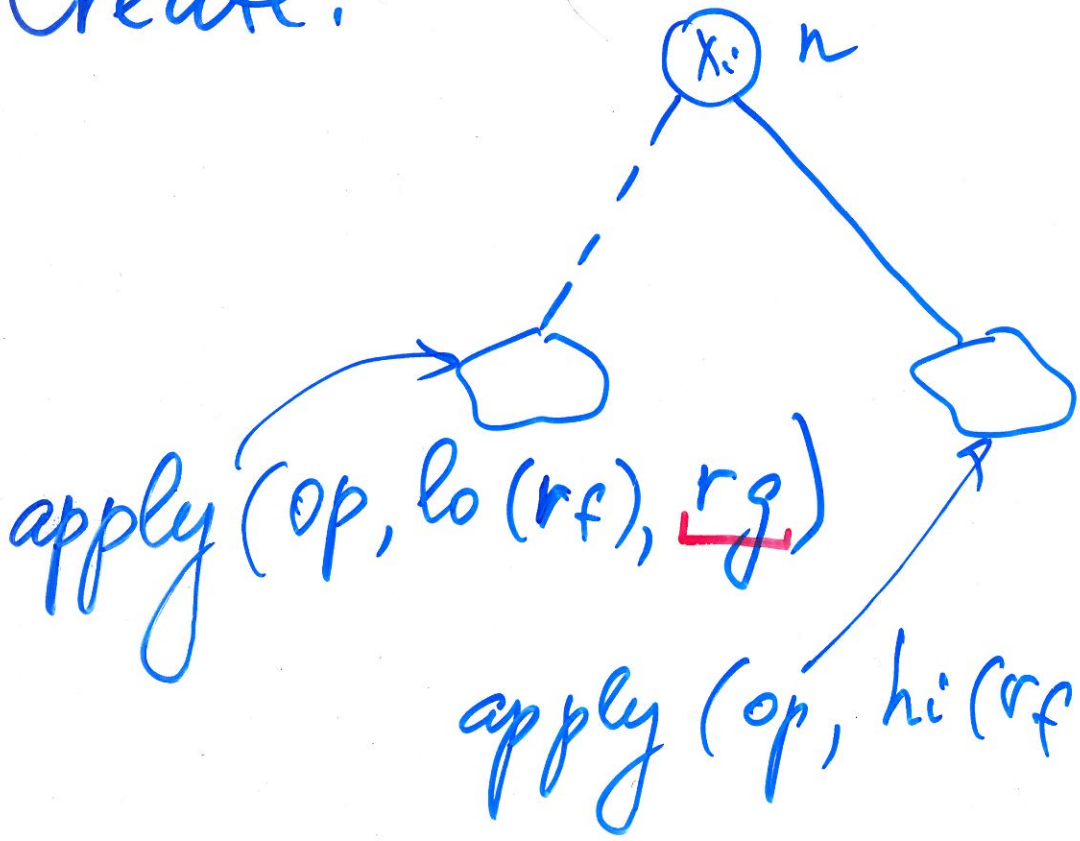
$j > i$

$\Rightarrow$ B_g does not have an x_i-node

$\Rightarrow$ g is independent of x_i:

$(g = g[0/x_i] = g[1/x_i])$

Create:



Finish by calling reduce.

$O(2 \cdot |B_f| \cdot |B_g|)$

(with memoisation)

Restrict

Computes $f[0/x]$, $f[1/x]$

for $f[0/x]$:

for each node n labelled with x, incoming edges are redirected to $lo(n)$

for $f[1/x]$ similar

— " — to $hi(n)$

Algorithm Exists

Define:

$$\exists x.f \text{ as } f[0/x] + f[1/x]$$

Intuition: $\exists x.f$ is true if f could be made true by choosing the value for x to be either 0 or 1

Thus, **Exists** can be implemented as

apply(+, restrict(0, x, B_f), restrict(1, x, B_f))

(It is possible to improve on this algorithm)