Intelligent Systems Programming

Lecture 7: **BDD Construction and Manipulation**

1. BDD construction
2. Boolean operations on BDDs
3. BDD-Based configuration

Today’s Program

- [10:00-10:50]
  - Unique table
  - Build(t)
  - Apply(op,u_1,u_2)
- [11:00-11:50] Configit A/S, Configuration

Henrik Hulgaard • 1st
CTO at Configit A/S
Configit • University of Washington

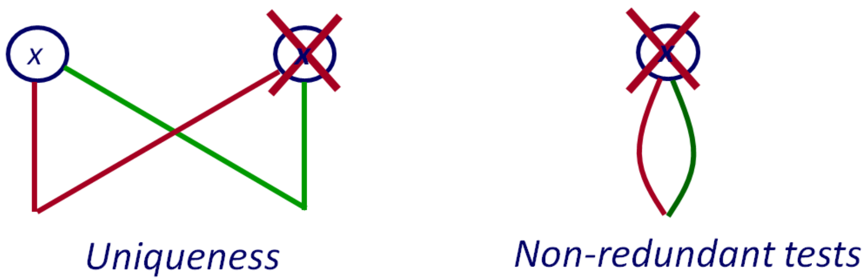
Henrik Reif Andersen • 1st
CEO, Configit A/S
Configit • Aarhus University

BDD Construction

BDD construction

Last week:

1. Make a Decision Tree of the Boolean expression
2. Keep reducing it until no further reductions are possible



This week:

- Reduce the decision tree to a BDD while building it

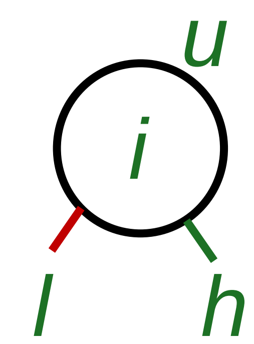
Reduce decision tree to BDD during construction

- Represent BDD by a table of unique nodes (UT)
- Build BDDs recursively,
  i.e. to add a new node u:
  1. Compute $high(u)$ and $low(u)$ and store them in UT
  2. Maintain BDD reductions when adding u to UT:
     a) Only extend UT with u if
        $high(u) \neq low(u)$ (non-redundancy test)
     b) Only extend UT with u if $u \notin UT$
        (uniqueness)

Unique Table Representation

Node Attributes

u	unique node identifier $\{\mathbf{0},\mathbf{1},2,3,\ldots\}$
i	variable index $\{1,2,\ldots,n,\mathbf{n+1}\}$
l	node identifier of low
h	node identifier of high



Represent Unique Table by two tables *T* and *H*

$T : u \rightarrow (i,l,h)$

$H: (i,l,h) \rightarrow u$

H is the inverse of T:

$T(u) = (i,l,h) \Leftrightarrow H(i,l,h) = u$

Primitive Operations on T and H

$T : u \mapsto (i, l, h)$

$init(T)$	initialize T to contain only 0 and 1
$u \leftarrow add(T, i, l, h)$	allocate a new node u with attributes (i, l, h)
$var(u), low(u), high(u)$	lookup the attributes of u in T

$H : (i, l, h) \mapsto u$

$init(H)$	initialize H to be empty
$b \leftarrow member(H, i, l, h)$	check if (i, l, h) is in H
$u \leftarrow lookup(H, i, l, h)$	find $H(i, l, h)$
$insert(H, i, l, h, u)$	make (i, l, h) map to u in H

Unique Table Interface: MakeNode (Mk)

$\textsc{Mk}[T, H](i, l, h)$

$\quad$ **if** $l = h$ **then return** l

$\quad$ **else if** $member(H, i, l, h)$ **then**

$\qquad$ **return** $lookup(H, i, l, h)$

$\quad$ **else** $u \leftarrow add(T, i, l, h)$

$\qquad$ $insert(H, i, l, h, u)$

$\qquad$ **return** u

Let’s do example on T, H and Mk!

Build

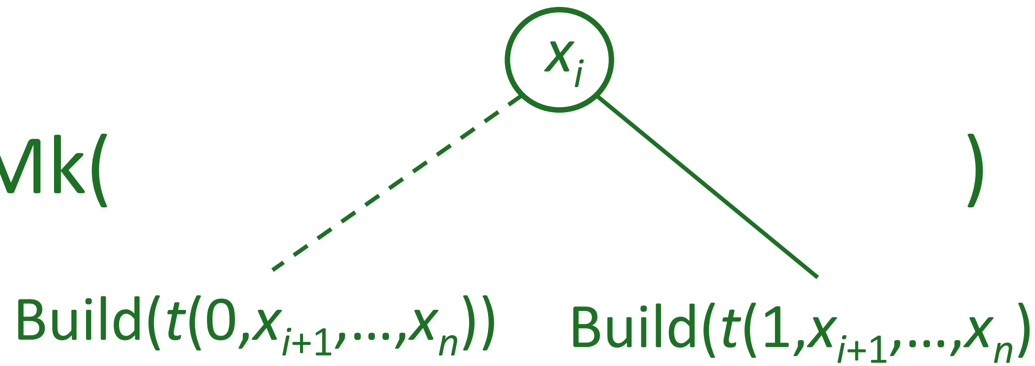
Idea: Construct the BDD recursively using the Shannon Expansion $t = x \rightarrow t[1/x], t[0/x]$

- Terminal cases

  $\text{Build}(0) = \boxed{0}$

  $\text{Build}(1) = \boxed{1}$

- Recursive case

  $\text{Build}(t(x_i, x_{i+1}, \ldots, x_n)) = \text{Mk}($  $)$

Build

$\textsc{Build}[T, H](t)$

function $\textsc{build}'(t, i) =$

  if $i > n$ **then**

    if t is false **then return** 0 **else return** 1

  else $v_0 \leftarrow \textsc{build}'(t[0/x_i], i+1)$

    $v_1 \leftarrow \textsc{build}'(t[1/x_i], i+1)$

    return $\textsc{mk}(i, v_0, v_1)$

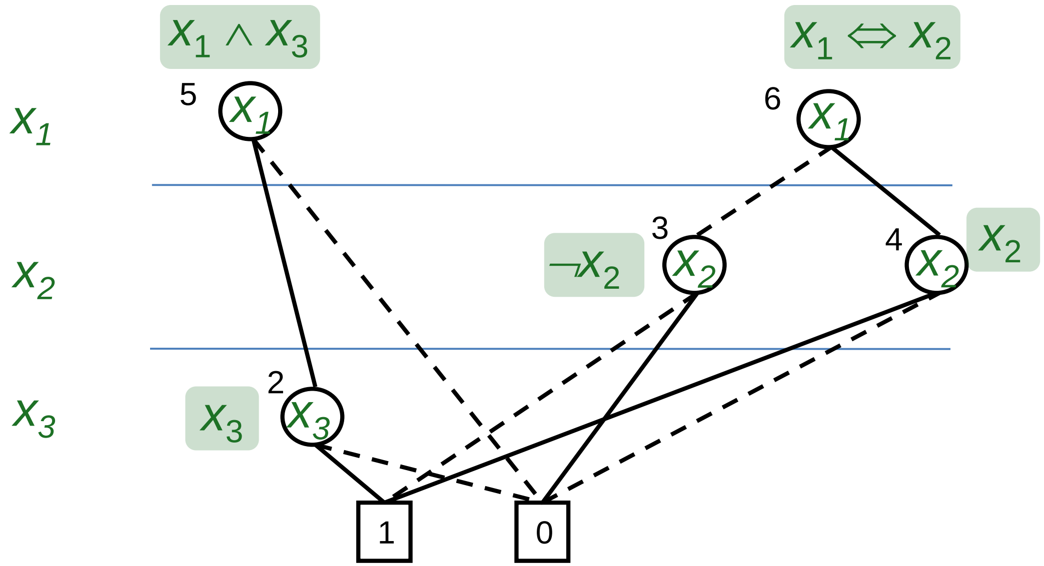
end $\textsc{build}'$

return $\textsc{build}'(t, 1)$

BDD Manipulation

Multi-Rooted BDD

Unique Table contains many BDDs



Apply

- Apply(op, u_1, u_2): computes the BDD of

$$u_1 \; op \; u_2$$

where

op : any of the 16 Boolean operators

u_1, u_2: root nodes of BDDs

- Relies on the Shannon expansion properties:

$$(x \to t_1, t_0) \; op \; (x \to t'_1, t'_0) \;\equiv\; x \to (t_1 \; op \; t'_1), (t_0 \; op \; t'_0)$$

$$(x \to t_1, t_0) \; op \; t \;\equiv\; x \to (t_1 \; op \; t), (t_0 \; op \; t)$$

Apply with $op = \wedge$

- **Terminal case:** $u \in \{0,1\}$
  $u' \in \{0,1\}$

  App($u \wedge u'$) $= u \wedge u'$

- **Recursive case:** $u = x_v \rightarrow u_1, u_0$
  $u' = x_w \rightarrow u'_1, u'_0$

  App($u \wedge u'$) $=$

  Mk(x_v, App($u_0 \wedge u'_0$), App($u_1 \wedge u'_1$)) if $v = w$

  Mk(x_v, App($u_0 \wedge u'$), App($u_1 \wedge u'$)) if $v < w$

  Mk(x_w, App($u \wedge u'_0$), App($u \wedge u'_1$)) if $w < v$

```
APPLY[T, H](op, u1, u2)
  init(G)

  function APP(u1, u2) =
    if G(u1, u2) ≠ empty then return G(u1, u2)
    else if u1 ∈ {0, 1} and u2 ∈ {0, 1} then u ← op(u1, u2)
    else if var(u1) = var(u2) then
        u ← MK(var(u1), APP(low(u1), low(u2)), APP(high(u1), high(u2)))
    else if var(u1) < var(u2) then
        u ← MK(var(u1), APP(low(u1), u2), APP(high(u1), u2))
    else (* var(u1) > var(u2) *)
        u ← MK(var(u2), APP(u1, low(u2)), APP(u1, high(u2)))
    G(u1, u2) ← u
    return u
  end APP

  return APP(u1, u2)
```

Properties of Apply

- Improvements?
  - Early termination. E.g., no reason to keep recursing if the left side in a conjunction is 0

- Complexity : $O(|u_1||u_2|)$, due to dynamic programming

- So a BDD of any formula can be computed in poly time?

Construct BDDs from expression tree