CS357 Lecture: BDD basics

David Dill

BDDs (Boolean/binary decision diagrams)

BDDs are a very successful representation for Boolean functions.

A BDD represents a Boolean function on variables x_1, x_2, ... x_n.

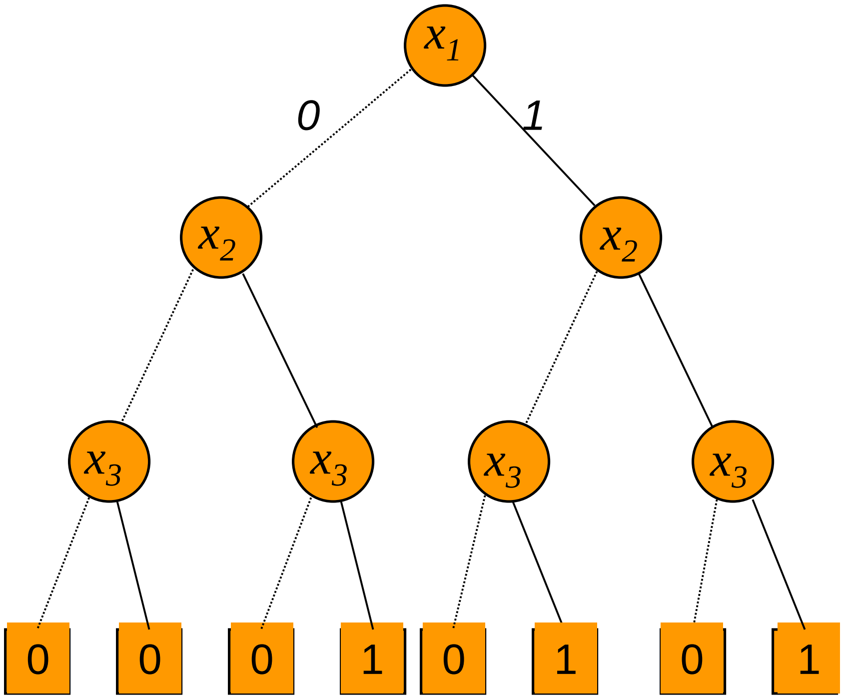
BDDs have many applications in CAD for digital systems, and formal verification (and current context-sensitive analysis, according to Alex).

Boolean functions are fundamental to computation, so there are many other applications as well.

Don Knuth: "one of the only really fundamental data structures that came out in the last twenty-five years"

Ordered decision tree

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

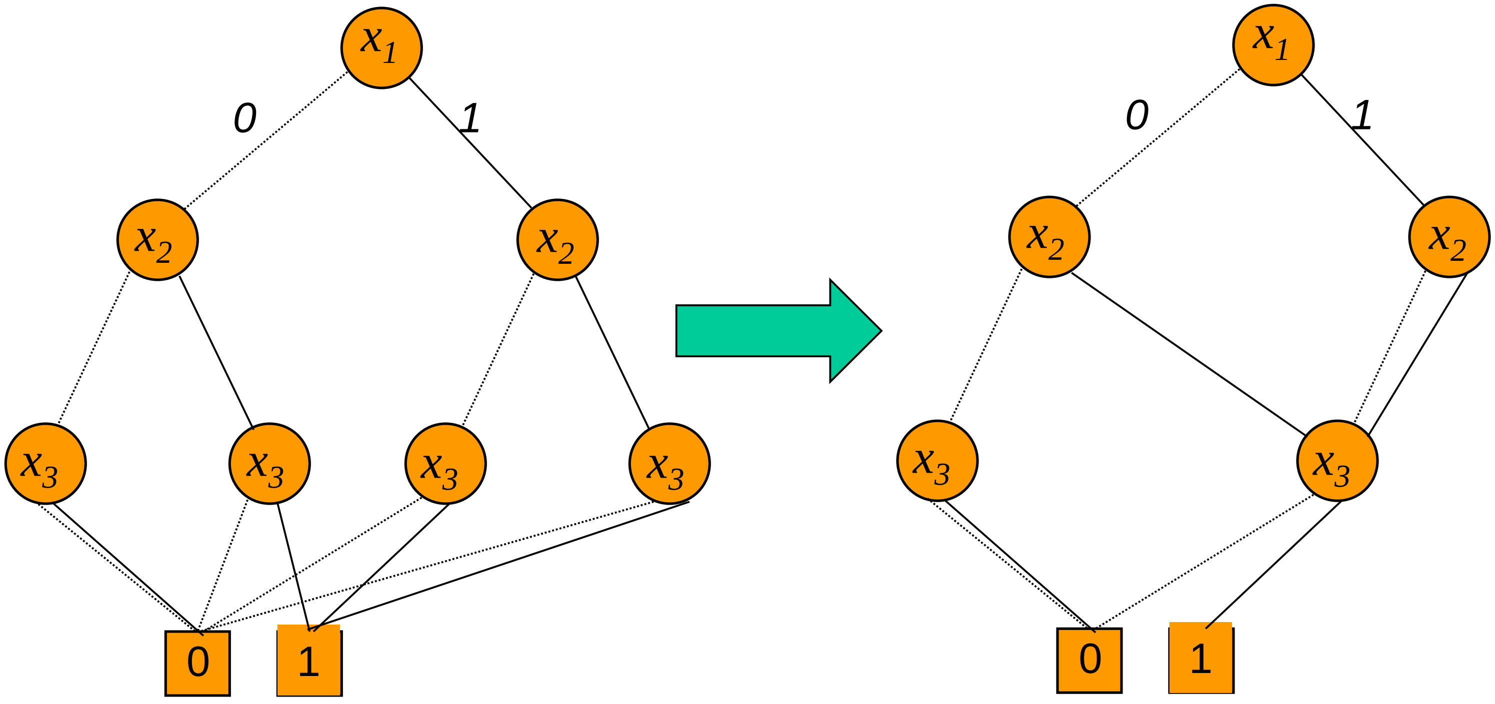


Crucial constraint:

There is a global total order on the variables.

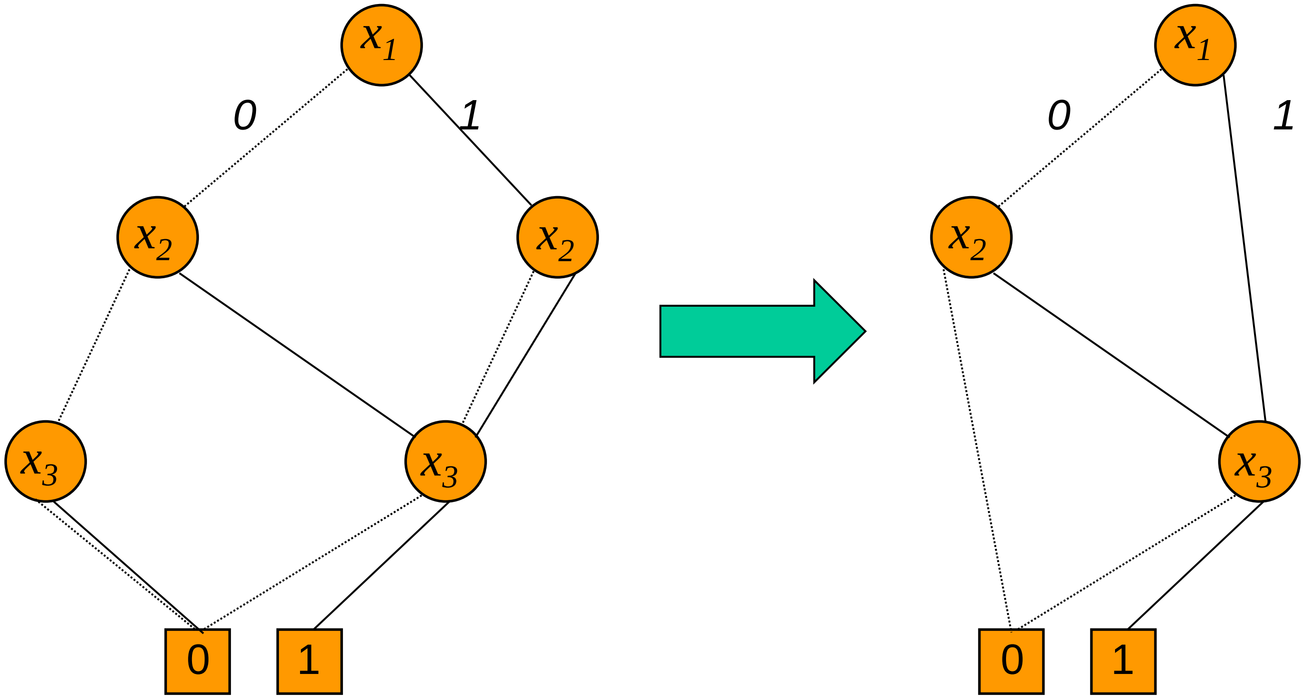
Variables appear in this order along all paths in the BDD.

Space optimization 1: share identical subtrees

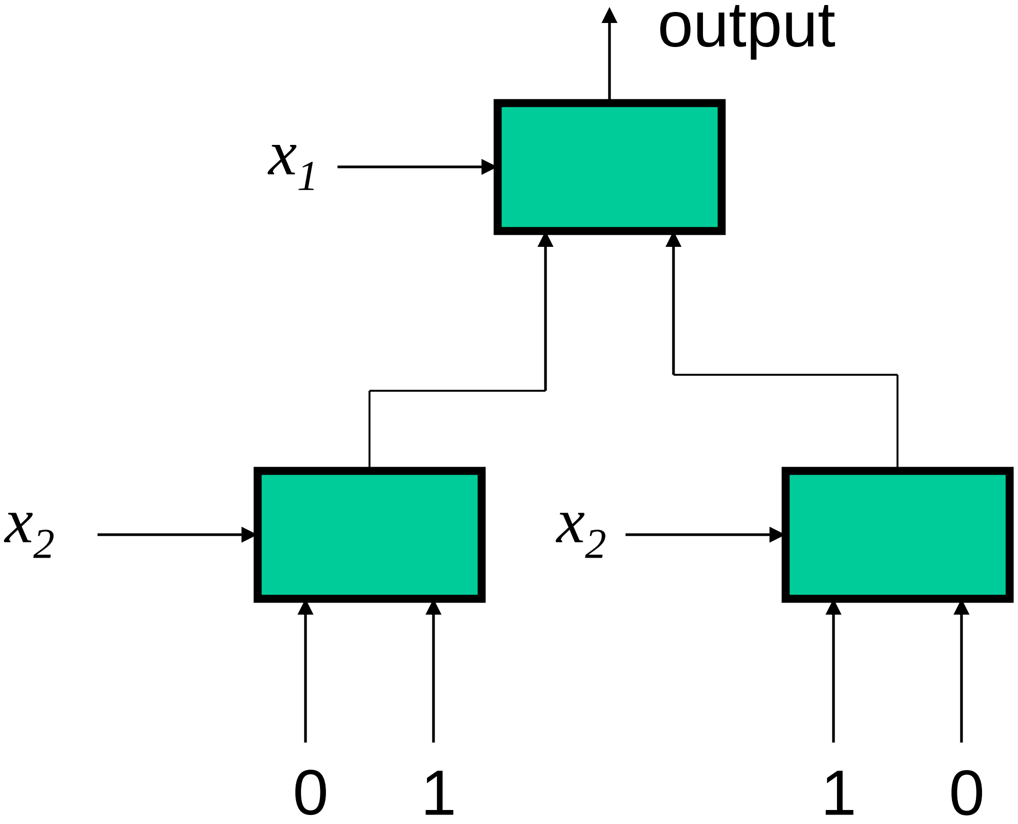


Nodes can be merged bottom-up

Space optimization 2: delete nodes with identical children



Multiplexer tree interpretation



BDDs can be regarded as digital circuits

Canonicity of BDDs

A BDD is a canonical representation of a Boolean function.

- There is only one BDD for a Boolean function
- Boolean expressions are *not* canonical:

  $(a \wedge b) \vee (a \wedge c)$ is same as $(a \wedge (b \vee c))$

- Easy to check for
  - Boolean equivalence (same BDD)
  - Tautology (= 1)
  - Unsatisfiability (= 0)

Implementation

BDD data structure is either "0", "1" or pointer to a record with *var*, *thenbdd*, *elsebdd*.

BDDs are constructed bottom-up.

For subtree sharing, maintain a hash table that maps

<*var*, *thenptr*, *elseptr*> to *bddptr*

Implementation

Assumes var < any vars appearing in `thenbdd, elsebdd`

```
make_bdd(var, thenbdd, elsebdd):
 if thenbdd = elsebdd return thenbdd;
 else if (lookup(var, thenbdd, elsebdd))
     return old bdd from hash table;
 else {
   b = new_node(var, thenbdd, elsebdd);
   inserthash(<var, thenbdd, elsebdd>, b);
   return b;
 };
```

Cofactors and Shannon decomposition

Cofactoring a Boolean function restricts the function to a particular value of a variable.

- $f \downarrow x_i$ is the cofactor of f when $x_i = 1$.
- This is the function you get if you replace x_i by 1 everywhere in an expression for f
- $f \downarrow \neg x_i$ is the cofactor of f when $x_i = 0$.

x_i does not appear in $f \downarrow x_i$ or in $f \downarrow \neg x_i$

Shannon decomposition

Allows recursive decomposition of functions

$$f = ite(x_i\ ,_,\ (f \downarrow x_i), (\mathrm{f} \downarrow \neg x_i))$$

This identity holds *regardless of where* x_i *appears in the variable order*,

but Shannon decomposition is *a lot faster* when *applied to the top-most variable*.

Logical operations on BDDs.

All propositional connectives are implemented in the following way (many other operations are, too).

1. Use Shannon decomposition to define a recursive function (on trees).
2. Save and reuse intermediate results (“memo-ize”).

Logical operations on BDDs.

```
AND(b1, b2):
 if b1 = 0 or b2 = 0 return 0;
 else if b1 = 1 return b2;
 else if b2 = 1 return b1;
 else if b1.var = b2.var
   return make_bdd(b1.var, AND(b1.t, b2.t), AND(b1.e, b2.e));
 else if b1.var < b2.var
   return make_bdd(b1.var, AND(b1.t, b2), AND(b1.e, b2));
 else if b2.var < b1.var
   return make_bdd(b2.var, AND(b1, b2.t), AND(b1, b2.e));
```

Logical operations on BDDs.

However, time to do this is proportional to tree size, which might be exponentially more than DAG size.

So, use dynamic programming ("memo-ization").

```
AND(b1, b2):
       ....
    if lookup_in_AND_table(b1, b2)
      return old value;
    else
      build new BDD "b"
      insert_in_AND_table(<b1, b2>, b);
```

Logical operations on BDDs.

After this optimization, cost is proportional to *product* of BDD sizes.

Same approach can be used for OR, NAND, XOR, etc.

Also, for ite(b1, b2, b3) - "if then else".

Logical operations on BDDs.

Translation from a logical expression to a BDD:

Bottom-up evaluation of expression, using BDD operations.

```
BDD_trans(f):
 if f is a variable,
   build and return the BDD for it.
 else if f = g & h,
   return
     AND(BDD_trans(g), BDD_trans(h));
 etc.
```

BDD size

- Conversion from expression to BDD can cause exponential blowup
  - Each logical operation can result in a BDD that is product of input BDDs (details depend on how you measure size).
  - *n* operations on BDD's of size 2: $O(2^k)$
  - Any smaller result would have exciting implications for NP completeness.
  - Sometimes, BDD's are small

    Usually, you have to work at it!

BDD size

BDD size is strongly influenced by variable order

- BDD's are almost never small unless variable order is good.

Some functions always have small BDDs

- Symmetric functions (order of inputs doesn't matter).
- Addition
- Bitwise equivalence

"Most" functions have large BDDs

- Multiplication (any output bit).
- FFT, division, . . .

Variable order

- A good variable order is usually essential for compact BDDs
- Static variable ordering: Use heuristics to find a good order, i.e.
  - Interleave bits of operands of operators
  - Operand-specific orders (e.g., addition low-order to high-order or vice versa)
  - Put “control variables” at the top of the order (e.g. variables in conditionals that totally change functional behavior).

Dynamic Variable Ordering

- Variable ordering can also be done dynamically as BDD operations proceed.
- Optimal variable order problem is NP-complete.
- Many heuristics proposed. Rudell's "sifting" is widely used.
  - Try moving a variable to all other positions, leaving the others fixed. Then place variable in the position that minimizes BDD size.
  - Do this for all variables.
- Sifting is enabled by fast "swapping" of two adjacent variables in a BDD.

Dynamic Variable Ordering

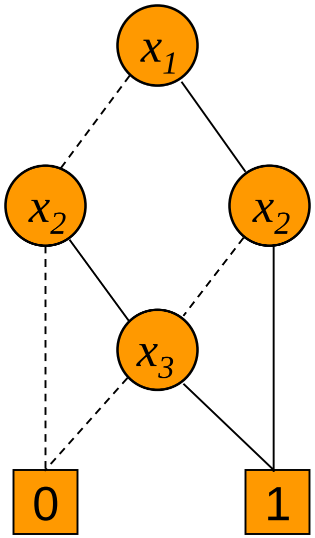
- Enabled a leap in BDD effectiveness.
- Can be SLOW. Feels like garbage collection (BDD operations stop while it reorders), but slower.
- Sometimes, for a particular problem, you can save the order found by sifting and reuse it effectively.

BDDs for symmetric functions

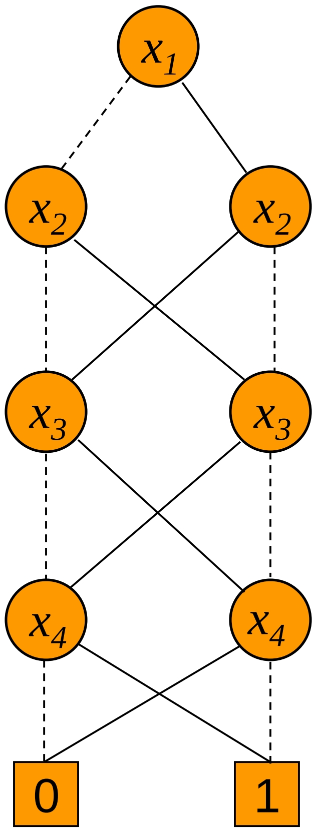
- symmetric: function is the same even if variables change.
- Examples:
  - AND(x1, x2, ..., xn) [OR, NAND, NOR]
  - PARITY(x1, x2, ..., xn) [n-ary XOR]
  - Threshhold functions (at least k out of n bits are 1)
- Size is same regardless of variable ordering (why?)

Symmetric function example

Majority function: output is majority value of three inputs.



Parity: 1 iff odd number of true variables.



Intuition: BDD nodes only need to “remember” number of variables that are 1.

Boolean quantification

$\exists x \,.\, f(x, y)$ is equivalent to "$f(x, y)$ holds for some value of x"

$\forall x \,.\, f(x, y)$ is equivalent to "$f(x, y)$ holds for all values of x"

Boolean quantifiers can be regarded as BDD operations:

$\exists x.\ f(x,y) = f\downarrow x \vee f\downarrow\neg x$

$\forall x.\ f(x,y) = f\downarrow x \wedge f\downarrow\neg x$

Boolean Quantification Implementation

BDD_exists(*x*, *f*) :

```
if (f.var != x) then
 make_bdd(f.var,
          BDD_exists(x, b.t),
          BDD_exists(x, b.e))
else
 bdd_OR(b.t, b.e));
```

(but memo-ize, of course)

forall(*x*,*f*) is similar

Predicates and Relations

For many applications, it is more convenient to use "word-level" notation.

For this, we can use "vector variables", which are fixed-length vectors of BDD variables (notation: $\boldsymbol{x}$)

Bit-vector expressions can be represented as vectors of BDDs, $F(\boldsymbol{x})$

This is just like bit-blasting with SAT.

Predicates and Relations

A binary relation on *Q* is a subset of *Q x Q*.

We can represent by creating two copies of the variables: ***x*** ("present state vars) and ***x'*** ("next state vars").

Then BDD *R(**x**, **x'**)* can represent the relation.

Notation**: P(**x**)* -- predicate on vectors of free variables (BDD vars) ***x returns a single-bit value.

Predicates and Relations

Example relation: $\boldsymbol{x} = \boldsymbol{x}'$, where $\boldsymbol{x}$ is a vector of Boolean variables:

$\boldsymbol{x} = \boldsymbol{x}'$ is $\bigwedge_{1 \leq i \leq n} (x_i \leftrightarrow x_i')$

"$\leftrightarrow$" is "iff" = "not xor"

Fake First-order logic

- Use vectors variables and vectors of BDDs.
- Functions, predicates, and quantifiers can be defined on vectors.
- Universal and existential quantifiers
- Not really first-order because vectors are fixed-length
  - all variable and expression types are finite

Operations on BDD vectors

Bitwise logical operations: e.g. $\boldsymbol{a} \oplus \boldsymbol{b}$

a, ***b*** must be of same length

Result is bitvector of XORs of corresponding bits from ***a*** and ***b***.

Same idea can be used for all other Boolean operators: AND, OR, NAND, etc.

Predicates

- Def: *Support* of a Boolean function is set of variables that can affect value (i.e. vars appearing in BDD).
- Notation: $P(\boldsymbol{x},\boldsymbol{y})$: $\boldsymbol{x},\boldsymbol{y}$ are vectors of variables containing superset of support of P.
- $P(\boldsymbol{y})$ is $P(\boldsymbol{x})$ with $\boldsymbol{y}$ variables substituted for corresponding $\boldsymbol{x}$ variables.
- Equality: $\boldsymbol{x} = \boldsymbol{y}$: $(x_0=y_0)\wedge(x_1=y_1)\wedge\ldots\wedge(x_n=y_n)$

Quantifiers

- Boolean quantification operations (previous lecture) can be extended to vectors of variables.
- $\exists \boldsymbol{x}.P(\boldsymbol{x},\boldsymbol{y}) = \exists x_0 \exists x_1 \ldots P(x_0, x_1, \ldots, y_0, y_1, \ldots)$
- Existential quantification is *projection*

$P(x,y)$

x_0	x_1	y_0	y_1	y_2
0	0	0	0	0
0	0	0	1	0
0	1	1	0	0
0	1	1	1	1
1	1	0	0	0

$\exists x.P(x,y)$

y_0	y_1	y_2
0	0	0
0	1	0
1	0	0
1	1	1
0	0	0

y_0	y_1	y_2
0	0	0
0	1	0
1	0	0
1	1	1

Variable renaming

Variables in BDD predicate can be *renamed* using the equality relation and existential quantification.

$$P(\boldsymbol{y}) = \exists \boldsymbol{x}.\ (\boldsymbol{x} = \boldsymbol{y}) \wedge P(\boldsymbol{x})$$

Note that this makes logical sense (when should $P(\boldsymbol{y})$ hold?), but also describes BDD operations.

Symbolic Breadth-First Search

- Search a graph without explicitly representing any vertices.
- $G = \langle V, E \rangle$
- V represented with bitvectors of sufficient length.
- E is a binary relation in $V \times V$
- **Reachability**: Find the set of all vertices that are reachable from an arbitrary vertex i.

Symbolic breadth first search

- Encode graph. For convenience, consider*V* be the set of all bitvectors of length *k.*
- *E(**x**, **y**)* holds if there is an edge from ***x** to **y**.*
- *I(**x**)* says *"**x*** is an initial vertex.*"*
- Reachability:
  - $R_0(\boldsymbol{x}) = I(\boldsymbol{x})$
  - $R_{n+1}(x) = R_n(\boldsymbol{x}) \vee \exists \boldsymbol{y}\ (\boldsymbol{x}=\boldsymbol{y} \wedge \exists \boldsymbol{x}\ [\ R_n(\boldsymbol{x}) \wedge R(\boldsymbol{x},\boldsymbol{y})\ ])$
  - Continue until $R_{n+1}(x) = R_n(\boldsymbol{x})$ = all reachable states.
- Computing the last thing (the "image computation") efficiently is *the central computational challenge* in many verification problems.