CS 243

Lecture 13

Datalog and BDD examples

1. Datalog example and walkthrough
2. BDD algorithms

Readings: Chapter 12

Example program

```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

void m(C a, C b) {
  n(a,b);
}

void n(C c, C d) {
  c.f = d;
}
```

Pointer Analysis in Datalog

Domains

- V = variables
- H = heap objects
- F = fields

EDB (input) relations

- $vP_0(v{:}V,h{:}H)$: object allocation sites
- $assign(v_1{:}V,v_2{:}V)$: assignment instructions ($v_1 = v_2;$) and parameter passing
- $store(v_1{:}V,f{:}F,v_2{:}V)$: store instructions ($v_1.f = v_2;$)
- $load(v_1{:}V,f{:}F,v_2{:}V)$: load instructions ($v_2 = v_1.f;$)

IDB (computed) relations

- $vP(v{:}V,h{:}H)$: variable points-to relation (v can point to object h)
- $hP(h_1{:}H,f{:}F,h_2{:}H)$: heap points-to relation (object h_1 field f can point to h_2)

Rules

- $vP(v,h)$:- $vP_0(v,h)$.
- $vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
- $hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
- $vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Step 1: Assign numbers to elements in domain

```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

void m(C a, C b) {
  n(a,b);
}

void n(C c, C d) {
  c.f = d;
}
```

Domains

V
- 'x' : 0
- 'y' : 1
- 'z' : 2
- 'a' : 3
- 'b' : 4
- 'c' : 5
- 'd' : 6

H
- 'main@1' : 0
- 'main@2' : 1
- 'main@3' : 2

F
- 'f' : 0

Step 2: Extract initial relations (EDB) from program

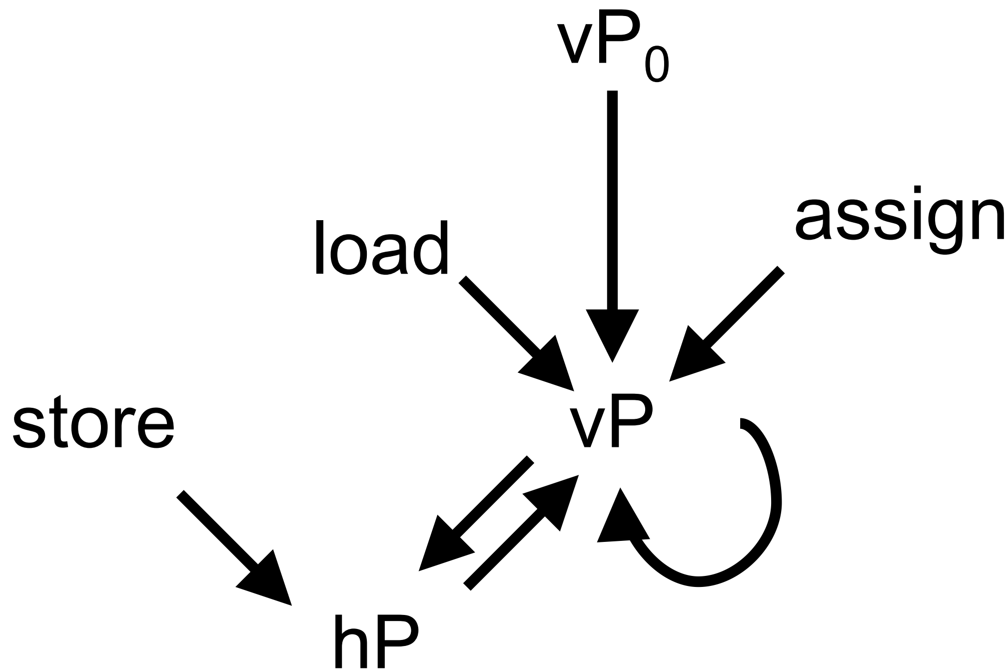
```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

void m(C a, C b) {
  n(a,b);
}

void n(C c, C d) {
  c.f = d;
}
```

vP_0('x', 'main@1').
vP_0('y', 'main@2').
vP_0('z', 'main@3').
assign('a','x').
assign('b','y').
assign('c','z').
assign('d','x').
load('z','f','q').
assign('c','a').
assign('d','b').
store('c','f','d').

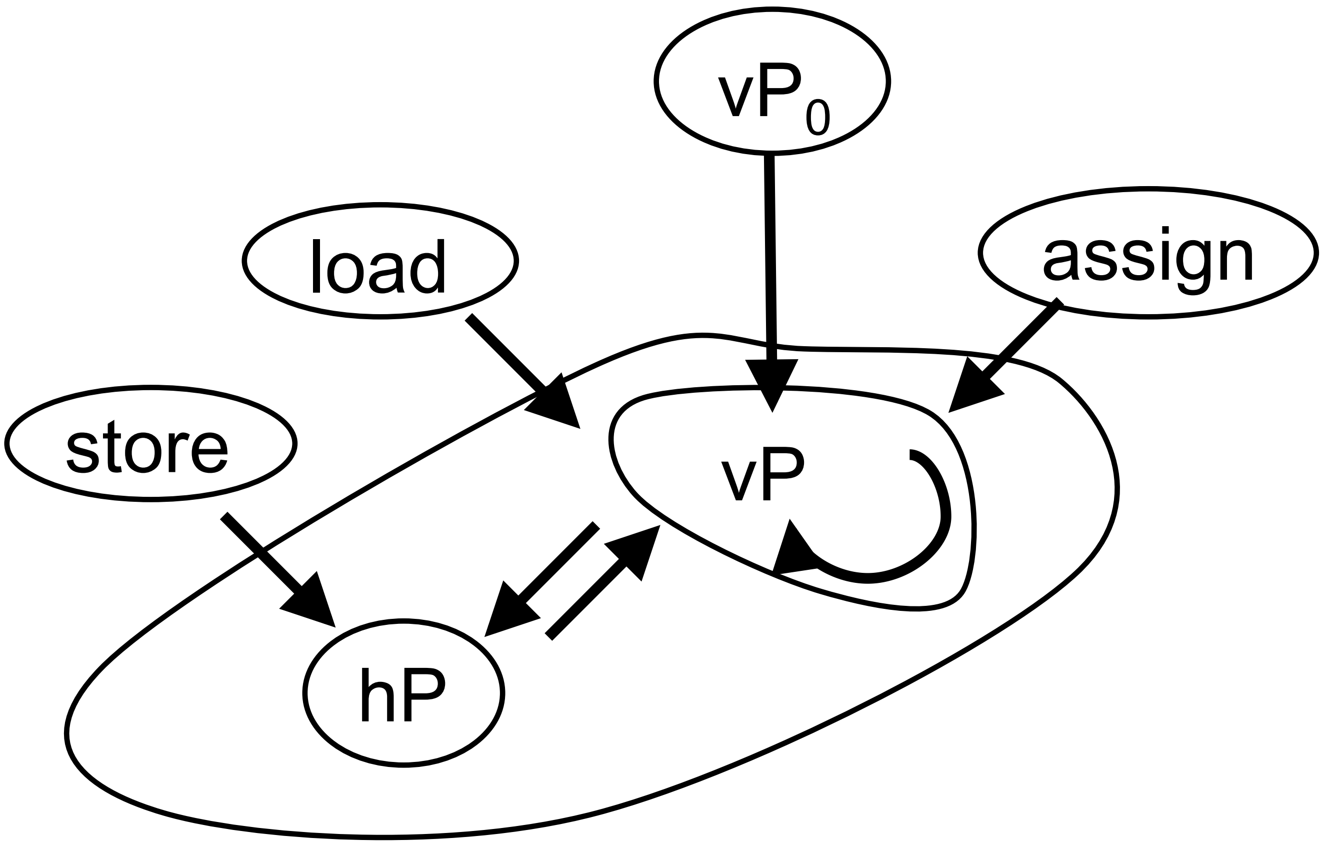
Step 3: Generate Predicate Dependency Graph

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.



Step 4: Determine Iteration Order



Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

hP

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').

hP

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a','x').
assign('b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').

hP

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').
vP('c','main@1').
vP('d','main@2').

hP

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').
vP('c','main@1').
vP('d','main@2').

hP

hP('main@1','f','main@1').
hP('main@1','f','main@2').
hP('main@3','f','main@1').
hP('main@3','f','main@2').

Step 5: Apply rules until convergence

Rules

$vP(v,h)$:- $vP_0(v,h)$.
$vP(v_1,h)$:- $assign(v_1,v_2)$, $vP(v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP(v_1,h_1)$, $vP(v_2,h_2)$.
$vP(v_2,h_2)$:- $load(v_1,f,v_2)$, $vP(v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

assign

assign('a’,'x').
assign(‘b','y').
assign('c','z').
assign('d','x').
assign('c','a').
assign('d','b').

vP

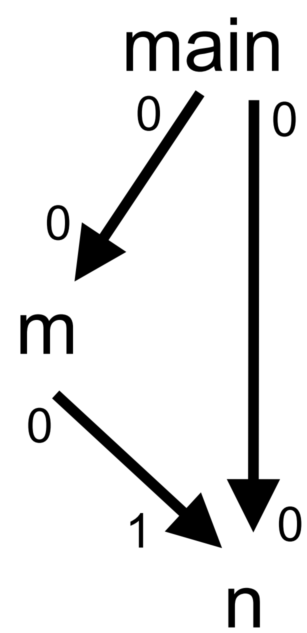
vP('x','main@1').
vP('y','main@2').
vP('z','main@3').
vP('a','main@1').
vP('d,'main@1').
vP('b','main@2').
vP('c','main@3').
vP('c','main@1').
vP('d','main@2').
vP('q','main@1').
vP('q','main@2').

hP

hP('main@1','f','main@1').
hP('main@1','f','main@2').
hP('main@3','f','main@1').
hP('main@3','f','main@2').

Context Numbering

```
void main() {
  x = new C();
  y = new C();
  z = new C();
  m(x,y);
  n(z,x);
  q = z.f;
}

void m(C a, C b) {
  n(a,b);
}

void n(C c, C d) {
  c.f = d;
}
```



Context-Sensitive Pointer Analysis

Domains

- C = context
- V = variables
- H = heap objects
- F = fields

EDB (input) relations

- $vP_0(v{:}V,h{:}H)$: object allocation sites
- $assign_C(c_1{:}C,v_1{:}V,c_2{:}C,v_2{:}V)$: context-sensitive assignments
- $store(v_1{:}V,f{:}F,v_2{:}V)$: store instructions ($v_1.f = v_2;$)
- $load(v_1{:}V,f{:}F,v_2{:}V)$: load instructions ($v_2 = v_1.f;$)

IDB (computed) relations

- $vP_C(c{:}C,v{:}V,h{:}H)$: context-sensitive variable points-to relation
- $hP(h_1{:}H,f{:}F,h_2{:}H)$: heap points-to relation (object h_1 field f can point to h_2)

Rules

- $vP_C(\_,v,h)$:- $vP_0(v,h)$.
- $vP_C(c_1,v_1,h)$:- $assign_C(c_1,v_1,c_2,v_2)$, $vP_C(c_2,v_2,h)$.
- $hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP_C(c,v_1,h_1)$, $vP_C(c,v_2,h_2)$.
- $vP_C(c,v_2,h2)$:- $load(v_1,f,v_2)$, $vP_C(c,v_1,h_1)$, $hP(h_1,f,h_2)$.

Apply context-sensitive rules until convergence

Rules

$vP_C(\_,v,h)$:- $vP_0(v,h)$.
$vP_C(c_1,v_1,h)$:- $assign_C(c_1,v_1,c_2,v_2)$, $vP_C(c_2,v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP_C(c,v_1,h_1)$, $vP_C(c,v_2,h_2)$.
$vP_C(c,v_2,h2)$:- $load(v_1,f,v_2)$, $vP_C(c,v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

$assign_C$

$assign_C$(0,'a’,0,'x').
$assign_C$(0,‘b',0,'y').
$assign_C$(0,'c',0,'z').
$assign_C$(0,'d',0,'x').
$assign_C$(1,'c',0,'a').
$assign_C$(1,'d',0,'b').

vP_C

hP

Apply context-sensitive rules until convergence

Rules

$vP_C(\_,v,h)$:- $vP_0(v,h)$.
$vP_C(c_1,v_1,h)$:- $assign_C(c_1,v_1,c_2,v_2)$, $vP_C(c_2,v_2,h)$.
$hP(h_1,f,h_2)$:- $store(v_1,f,v_2)$, $vP_C(c,v_1,h_1)$, $vP_C(c,v_2,h_2)$.
$vP_C(c,v_2,h2)$:- $load(v_1,f,v_2)$, $vP_C(c,v_1,h_1)$, $hP(h_1,f,h_2)$.

Relations

vP_0

vP_0('x','main@1').
vP_0('y','main@2').
vP_0('z','main@3').

store

store('c','f','d').

load

load('z','f','q').

$assign_C$

$assign_C$(0,'a’,0,'x').
$assign_C$(0,‘b',0,'y').
$assign_C$(0,'c',0,'z').
$assign_C$(0,'d',0,'x').
$assign_C$(1,'c',0,'a').
$assign_C$(1,'d',0,'b').

vP_C

vP_C(0,'x','main@1').
vP_C(0,'y','main@2').
vP_C(0,'z','main@3').
vP_C(0,'a','main@1').
vP_C(0,'d,'main@1').
vP_C(0,'b','main@2').
vP_C(0,'c','main@3').
vP_C(1,'c','main@1').
vP_C(1,'d','main@2').
vP_C(0,'q','main@1').
vP_C(0,'q','main@2').

hP

hP('main@3','f',
'main@1').
hP('main@1','f',
'main@2').

Datalog to Relational Algebra

$vP(v_1, o) \text{ :- } assign(v_1, v_2), vP(v_2, o).$

$t_1 = \rho_{variable \rightarrow source}(vP);$

$t_2 = assign \bowtie t_1;$

$t_3 = \pi_{source}(t_2);$

$t_4 = \rho_{dest \rightarrow variable}(t_3);$

$vP = vP \cup t_4;$

Incrementalization

$t_1 = \rho_{variable \rightarrow source}(vP);$

$t_2 = assign \bowtie t_1;$

$t_3 = \pi_{source}(t_2);$

$t_4 = \rho_{dest \rightarrow variable}(t_3);$

$vP = vP \cup t_4;$

$vP'' = vP - vP';$

$vP' = vP;$

$assign'' = assign - assign';$

$assign' = assign;$

$t_1 = \rho_{variable \rightarrow source}(vP'');$

$t_2 = assign \bowtie t_1;$

$t_5 = \rho_{variable \rightarrow source}(vP);$

$t_6 = assign'' \bowtie t_5;$

$t_7 = t_2 \cup t_6;$

$t_3 = \pi_{source}(t_7);$

$t_4 = \rho_{dest \rightarrow variable}(t_3);$

$vP = vP \cup t_4;$

Optimize into BDD operations

vP’’ = vP – vP’;

vP’ = vP;

assign’’ = assign – assign’;

assign’ = assign;

$t_1 = \rho_{variable \rightarrow source}(vP'')$;

$t_2 = assign \bowtie t_1$;

$t_5 = \rho_{variable \rightarrow source}(vP)$;

$t_6 = assign'' \bowtie t_5$;

$t_7 = t_2 \cup t_6$;

$t_3 = \pi_{source}(t_7)$;

$t_4 = \rho_{dest \rightarrow variable}(t_3)$;

$vP = vP \cup t_4$;

vP’’ = diff(vP, vP’);

vP’ = copy(vP);

t_1 = replace(vP’’,$_{variable \rightarrow source}$);

t_3 = relprod(t_1,assign,$_{source}$);

t_4 = replace(t_3,$_{dest \rightarrow variable}$);

vP = or(vP, t_4);

Physical domain assignment

vP'' = diff(vP, vP');

vP' = copy(vP);

t_1 = replace(vP'',$_{variable \rightarrow source}$);

t_3 = relprod(t_1,assign,$_{source}$);

t_4 = replace(t_3,$_{dest \rightarrow variable}$);

vP = or(vP, t_4);

vP'' = diff(vP, vP');

vP' = copy(vP);

t_3 = relprod(vP'',assign,V0);

t_4 = replace(t_3, V1→V0);

vP = or(vP, t_4);

- Minimizing renames is NP-complete
- Renames have vastly different costs
- Priority-based assignment algorithm

Other optimizations

- Dead code elimination
- Constant propagation
- Definition-use chaining
- Redundancy elimination
- Global value numbering
- Copy propagation
- Liveness analysis

Splitting rules

R(a,e) :- A(a,b), B(b,c), C(c,d), R(d,e).

Can be split into:

T_1(a,c) :- A(a,b), B(b,c).

T_2(a,d) :- T_1(a,c), C(c,d).

R(a,e) :- T_2(a,d), R(d,e).

Affects incrementalization, iteration.

Use “split” keyword to auto-split rules.

"Minimal" Solution?

A(1,2).
B(2,3).
C(x,y) :– A(x,y).
C(x,y) :– C(x,z),C(z,y).
D(x,y) :– B(x,y),¬C(x,y).

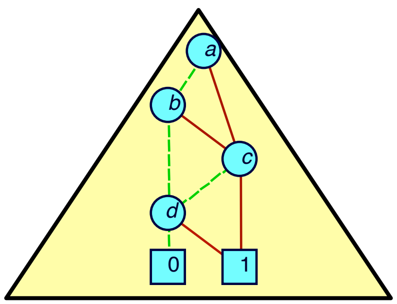
Solution 1: C(1,2), D(2,3)
Solution 2. C(1,2), C(2,3), C(1,3)

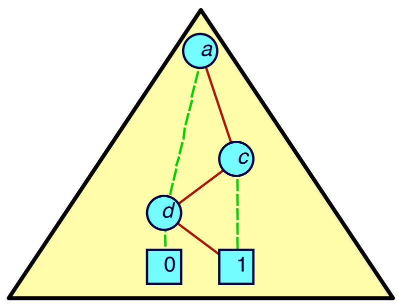
Which is preferable?

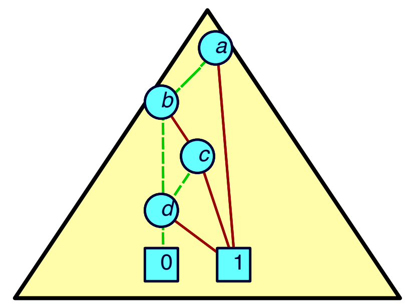
Apply Operation

- Concept
  - Basic technique for building OBDD from Boolean formula.

A *op* *B* *A op B*







Arguments *A*, *B*, *op*

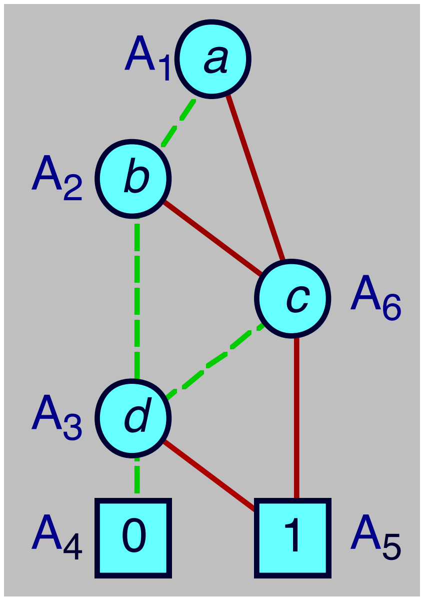
- ***A* and *B*: Boolean Functions**
  - **Represented as OBDDs**
- **op: Boolean Operation (e.g., ^, &, |)**

Result

- **OBDD representing composite function**
- *A op B*

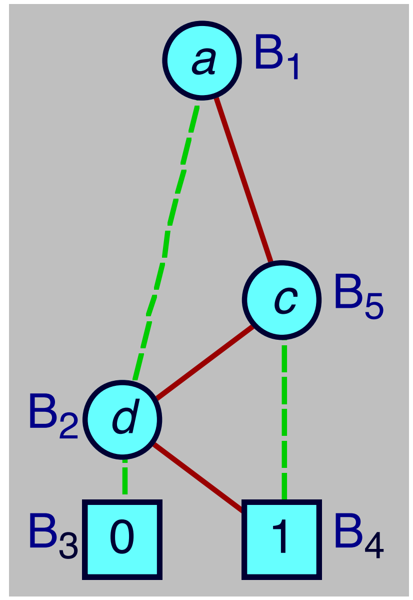
Apply Execution Example

Argument *A*

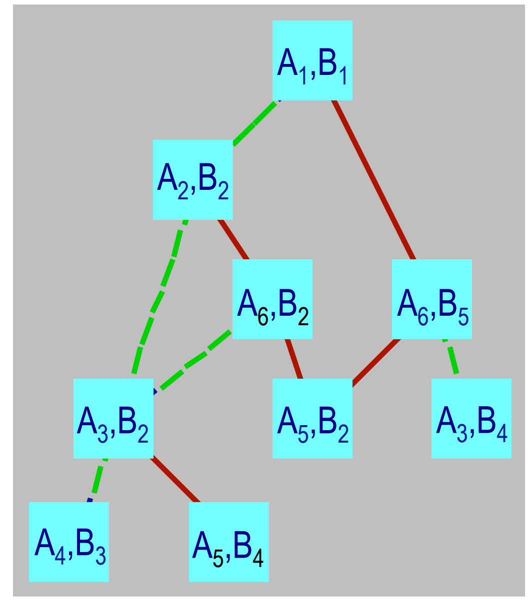


Operation

Argument *B*



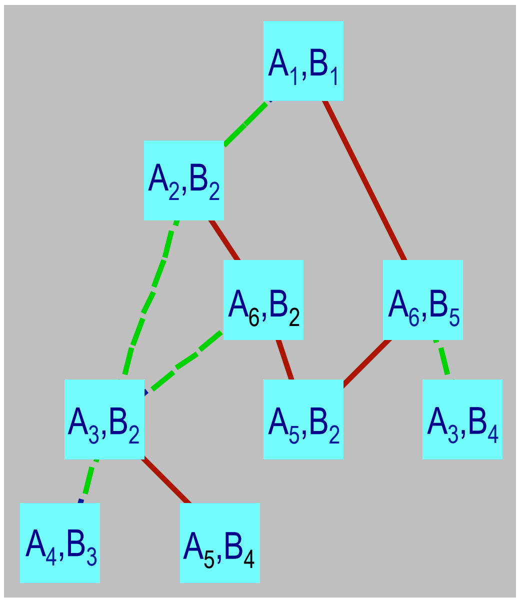
Recursive Calls



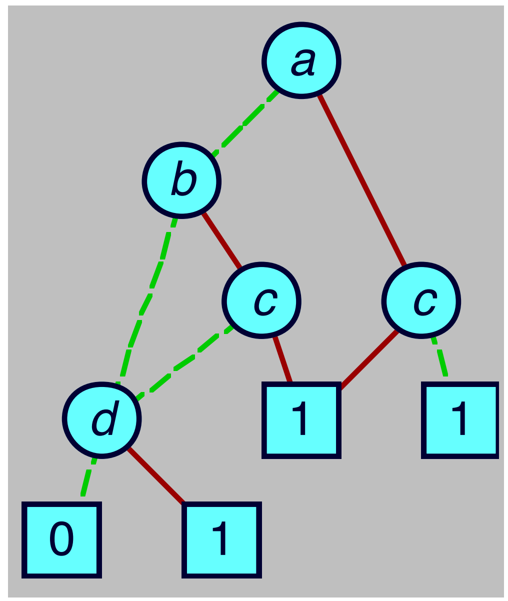
- Optimizations
  - Dynamic programming
  - Early termination rules

Apply Result Generation

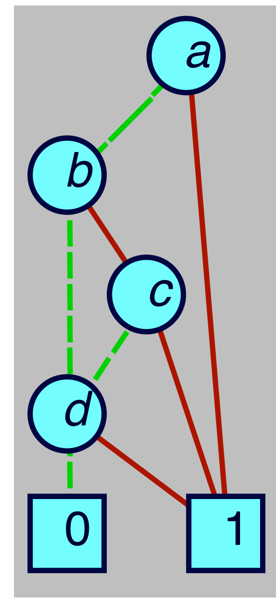
Recursive Calls



Without Reduction



With Reduction



- Recursive calling structure implicitly defines unreduced BDD
- Apply reduction rules bottom-up as return from recursive calls

BDD implementation

- ‘Unique’ table
  - Huge hash table
  - Each entry: level, left, right, hash, next
- Operation cache
  - Memoization cache for operations
- Garbage collection
  - Mark and sweep, free list.

Code for BDD 'and'.

Base case:

Memo cache lookup:

Recursive step:

Memo cache insert:

```
int and_rec(int l, int r) {
    BddCacheDataI entry;
    int res;

    if (l == r)
        return l;
    if (ISZERO(l) || ISZERO(r))
        return 0;
    if (ISONE(l))
        return r;
    if (ISONE(r))
        return l;
    entry = BddCache_lookupI(applycache, APPLYHASH(l, r, bddop_and));

    if (entry.a == l && entry.b == r && entry.c == bddop_and) {
        if (CACHESTATS)
            cachestats.opHit++;
        return entry.res;
    }
    if (CACHESTATS)
        cachestats.opMiss++;

    if (LEVEL(l) == LEVEL(r)) {
        PUSHREF(and_rec(LOW(l), LOW(r)));
        PUSHREF(and_rec(HIGH(l), HIGH(r)));
        res = bdd_makenode(LEVEL(l), READREF(2), READREF(1));
    } else if (LEVEL(l) < LEVEL(r)) {
        PUSHREF(and_rec(LOW(l), r));
        PUSHREF(and_rec(HIGH(l), r));
        res = bdd_makenode(LEVEL(l), READREF(2), READREF(1));
    } else {
        PUSHREF(and_rec(l, LOW(r)));
        PUSHREF(and_rec(l, HIGH(r)));
        res = bdd_makenode(LEVEL(r), READREF(2), READREF(1));
    }

    POPREF(2);

    entry.a = l;
    entry.b = r;
    entry.c = bddop_and;
    entry.res = res;

    return res;
}
```