

CIS 673: Lecture 7: Sept 27
BDD Implementation

- Vertices of all BDDs stored in a central pool
- Each BDD is simply an index into this global data structure
- Invariant property: no two vertices in the pool are isomorphic
- Advantages:
 - To check equivalence, compare indices
 - Sharing of subgraphs across nonisomorphic BDDs
- Performance depends on efficient garbage collection

BDD Data Structures

- Type of BDDs: **bdd**. Pointer to global data structure *BddPool*.

- Type of *BddPool*: **set of bddnode**

- Type of vertices: (*k* is number of variables)

$$\mathbf{bddnode} = ([1..k] \times \mathbf{bdd} \times \mathbf{bdd}) \cup \mathbf{bool}$$

- **bddnode** supports

- *Label*: **bddnode** \mapsto $[1..k]$

- *Left*: **bddnode** \mapsto **bdd**

- *Right*: **bddnode** \mapsto **bdd**

- *BddPool* has following operations

- Insert and IsMember

- *Index*: **bddnode** \mapsto **bdd**

- *BddPool*[*B*] returns vertex pointed to by *B*

Insertions into *BddPool*

- *BddPool* is initialized to contain two terminal vertices 0 and 1
- Add vertices only via *MakeVertex* to ensure that no vertices are isomorphic

function *MakeVertex*

Input: $i: [1..k]$, $B_0, B_1: \mathbf{bdd}$.

Output: $B: \mathbf{bdd}$ such that $r(B)$ equivalent to $(\neg x_i \wedge r(B_0)) \vee (x_i \wedge r(B_1))$.

begin

if $B_0 = B_1$ then return B_0 fi;

if $\neg \text{IsMember}((i, B_0, B_1), \text{BddPool})$ then

$\text{Insert}((i, B_0, B_1), \text{BddPool})$ fi;

return $\text{Index}((i, B_0, B_1))$

end.

Operations on BDDs

- Algorithms for boolean operations on BDDs
- Conjunction:
 - Input: Two pointers B_0 and B_1 into *BddPool*
 - Output: A pointer B such that the BDD represented by *BddPool*[B] should be the conjunction of the BDDs represented by *BddPool*[B_0] and *BddPool*[B_1]
- Recursive function *Conj* to take conjunction of vertices

Algorithm for Conjunction

```
function Conj
input  $B_0, B_1$  : bdd
output  $B$  : bdd
local  $v_0, v_1$  : bddnode;  $i, j$  :  $[1 \dots k]$ 
       $B, B_{00}, B_{01}, B_{10}, B_{11}$  : bdd
begin
   $v_0 := BddPool[B_0]$ ;
   $v_1 := BddPool[B_1]$ ;
  if  $v_0 = 0$  or  $v_1 = 1$  then return  $B_0$  fi;
  if  $v_0 = 1$  or  $v_1 = 0$  then return  $B_1$  fi;
   $i := Label(v_0)$ ;  $j := Label(v_1)$ ;
   $B_{00} := Left(v_0)$ ;  $B_{01} := Right(v_0)$ 
   $B_{10} := Left(v_1)$ ;  $B_{11} := Right(v_1)$ ;
  if  $i = j$  then
     $B := MakeVertex(i, Conj(B_{00}, B_{10}), Conj(B_{01}, B_{11}))$ 
  if  $i < j$  then
     $B := MakeVertex(i, Conj(B_{00}, B_1), Conj(B_{01}, B_1))$ 
  if  $i > j$  then
     $B := MakeVertex(j, Conj(B_0, B_{10}), Conj(B_0, B_{11}))$ 
  return  $B$ 
end.
```

Avoiding Recomputation

- Store computed results in a table *Done*
- Keys for *Done*: pairs of the form **bdd** \times **bdd**
- Stores value of the type **bdd**
- Initially empty
- Before recursive calls, *Conj* checks if *Done* $[(B_0, B_1)]$ has a value
- Upon termination *Conj* stores the result in *Done*

Modified Algorithm

```
function Conj
begin
   $v_0 := BddPool[B_0]; v_1 := BddPool[B_1];$ 
  if  $v_0 = 0$  or  $v_1 = 1$  then return  $B_0$  fi;
  if  $v_0 = 1$  or  $v_1 = 0$  then return  $B_1$  fi;
  if  $Done[(B_0, B_1)] \neq \perp$  then return  $Done[(B_0, B_1)]$ 
  if  $Done[(B_1, B_0)] \neq \perp$  then return  $Done[(B_1, B_0)]$ 
   $i := Label(v_0); j := Label(v_1);$ 
   $B_{00} := Left(v_0); B_{01} := Right(v_0)$ 
   $B_{10} := Left(v_1); B_{11} := Right(v_1);$ 
  if  $i = j$  then
     $B := MakeVertex(i, Conj(B_{00}, B_{10}), Conj(B_{01}, B_{11}))$ 
  if  $i < j$  then
     $B := MakeVertex(i, Conj(B_{00}, B_1), Conj(B_{01}, B_1))$ 
  if  $i > j$  then
     $B := MakeVertex(j, Conj(B_0, B_{10}), Conj(B_0, B_{11}))$ 
   $Done[(B_0, B_1)] := B;$ 
  return  $B$ 
end.
```

Operations on BDDs

- If B_0 contains m nodes and B_1 has n nodes then *Conj* has time complexity $O(mn)$
- Quadratic bound tight: conjunction of two BDDs with n nodes can have n^2 nodes (like finite automata)
- Similar algorithms for union, negation, simplification (i.e. computing $B[x := 0]$)
- Now we can translate a propositional formula to its BDD representation
- Translation can cause exponential blow-up

Multi-valued Decision Diagrams (MDD)

- Just like BDD, except choice need not be binary
- Variables have enumerated types
- Branching degree at an internal vertex x is k if x can take k values:
 - An internal vertex labeled with x has a successor edge labeled with m for every $m \in \mathbb{T}_x$
- Terminal vertices have 0 or 1 label as before
- Useful for verification of enumerated modules

Symbolic Invariant Verification

To solve the invariant verification problem (P, p)

1. Choose variable ordering over $X_P \cup X'_P$
2. Build the BDDs for
 - Initial region of P
 - Target region (negation of invariant p)
 - Transition relation of P
3. Execute symbolic search algorithm

Partitioned Transition Relation

- Transition relation is conjunction of predicates
- Do not compute the conjunction a priori, but only as needed (on-the-fly symbolic representation)
- A conjunctively-partitioned representation of a boolean expression p is a set $\{B_1, \dots, B_k\}$ of BDDs such that p is equivalent to the conjunction $r(B_1) \wedge \dots \wedge r(B_k)$
- The transition relation q^T is maintained as a set $\{q_1^T, \dots, q_k^T\}$ can have much smaller BDD representation
- In each iteration, if q is the current reachable region, we need to compute $q \wedge q^T$
- In each iteration, we need compute the conjunction $q \wedge q_1^T \wedge \dots \wedge q_k^T$

- This is typically much smaller than BDD for q^T : only reachable transitions are considered (the domain of various conjuncts q_i^T is constrained, or simplified by q)
- For us, for every atom, there is a conjunct representing update command of the atom
- Further partitioning possible: eg. each conjunct is a disjunction of guarded commands, maintain this in a disjunctively partitioned format

Early Quantifier Elimination

- If p does not depend upon x then

$$\exists x. p \wedge q \equiv p \wedge \exists x. q$$

- BDD for $\exists x. q$ is likely to be smaller than BDD of q
- In computing *PostReg*, we need to compute

$$\exists X. (q \wedge q_1^T \wedge \dots \wedge q_k^T)$$

- Strategy: take one conjunction at a time, eliminate (via existential quantification) as many variables as possible
- Quantifier elimination has a precedence over conjunction

Ordering of Conjuncts

- Effectiveness of early quantifier elimination depends on the ordering of conjuncts
- In 3-bit counter example, we want to compute

$$\exists\{out_0, out_1, out_2\}. (p \wedge q_0^T \wedge q_1^T \wedge q_2^T)$$

- If we compute $p \wedge q_0^T$ first, no existential quantification can be applied
- If we compute $p \wedge q_2^T$ first, out_2 can be eliminated
- Optimal computing strategy:

$$\exists out_0. (q_0^T \wedge \exists out_1. (q_1^T \wedge \exists out_2. (q_2^T \wedge p)))$$

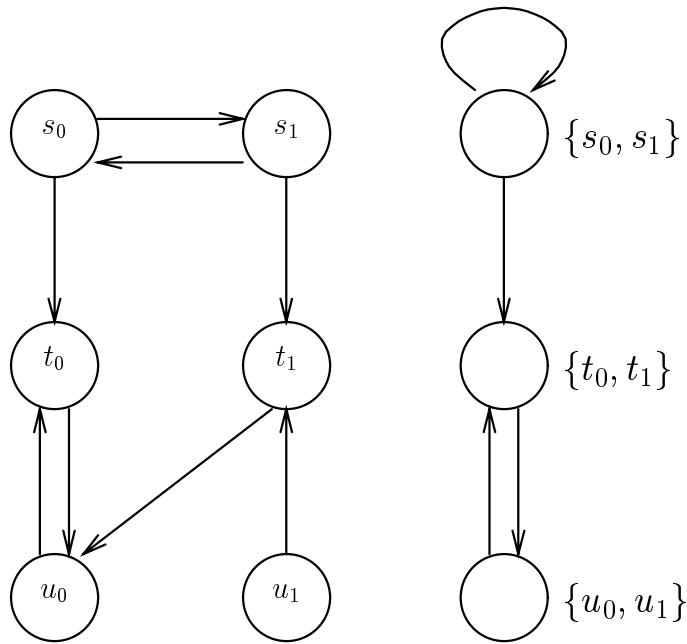
Battling State-Space Explosion

- On-the-fly search
- Partial-order reduction
- Symbolic search
- Abstractions based on
 - symmetries
 - eliminating variables
 - graph minimization

Graph Abstraction

- A partition \cong is an equivalence relation over state-space
- For a graph $G = (\Sigma, \sigma^I, \rightarrow)$ and partition \cong , the quotient G/\cong is a transition graph:
 - States: Equivalence classes of \cong
 - Initial states: σ is initial if $\sigma \cap \sigma^I$ is nonempty
 - Transitions: $\sigma \rightarrow_{\cong} \tau$ if there are two states $s \in \sigma$ and $t \in \tau$ such that $s \rightarrow t$
- Abstraction technique: choose a partition \cong and analyze the quotient graph G/\cong

Quotient Example



Invariant Verification

- Original reachability problem (G, σ^T)
- Solve $(G/\cong, \sigma^T/\cong)$, where σ^T/\cong contains classes that intersect with σ^T
- If modified problem says No, then the answer to the original problem is also No.
- If modified problem says Yes, then no conclusion about the original problem
- Abstraction is a sound, but incomplete, technique for checking invariants

Reachability-preserving Partitions

- Under which conditions quotients preserve the reachability properties?
- STABLE partition: The partition \cong is stable if $s \cong t$ and $s \rightarrow s'$, then there is a state t' such that $s' \cong t'$ and $t \rightarrow t'$
- Condition on target region: σ^T should be a block of \cong (union of equivalence classes)
- Theorem: The reachability problems (G, σ^T) and $(G/\cong, \sigma^T/\cong)$ have same answers if
 - \cong is stable
 - σ^T is a block of \cong

Examples of Stable Partitions

- Projection onto a subset $Y \subseteq X_P$ of module variables: $s \cong_{[Y]} t$ if $Y[s] = Y[t]$
- Projection onto latched variables is a stable partition
- Stable Variable Sets:
 - Closed under dependency
 - A subset X is stable if for every variable $x \in X$, if the variable x is controlled by the atom U of P then both $\text{read}X_U \subseteq X$ and $\text{await}X_U \subseteq X$.
- Projection onto a stable variable set is a stable partition
- Strategy for invariant verification (P, p) : Find the smallest stable variable set that contains all variables occurring in p

Symmetries

- If the system contains renamed copies of same module definition, then the state-space exhibits symmetries
- The processes P_1 and P_2 of Pete
- Two trains of the railroad controller
- Processes and buffers in agreement protocol
- Symmetric states: one can be obtained from the other by permuting indices
- Formalization via graph automorphisms

Graph Automorphisms

- Transition graph $G = (\Sigma, \sigma^I, \rightarrow)$.
- A G -automorphism is a bijection f from Σ to Σ such that
 - Initial states are mapped to initial states:
 $f(\sigma^I) = \sigma^I$
 - Preserves transitions: $s \rightarrow t$ iff $f(s) \rightarrow f(t)$.
- Identity function id is a G -automorphism
- Inverse of automorphism is also automorphism
- Functional composition of two automorphisms is an automorphism
- The subgroup generated by a set of automorphisms contains only automorphisms

Groups

- A group is a set A with a multiplication operation \circ :
 - \circ is associative,
 - there exists an element that is identity for \circ
 - every element has inverse with respect to \circ
- A subgroup of (A, \circ) is a subset $B \subseteq A$ such that (B, \circ) is a group
- For a subset $B \subseteq A$, the subgroup generated by B , denoted $\text{closure}(B)$, is the smallest subgroup of (A, \circ) that contains B
- The elements in B are called generators for the group $\text{closure}(B)$
- Example: the set of functions with functional composition

Symmetric Partition

- F : Set of G -automorphisms
- $\text{closure}(F)$: Subgroup generated by F
- \cong^F : two states s and t are equivalent if there exists $f \in \text{closure}(F)$ such that $t = f(s)$
- Theorem: \cong^F is stable
- If the target region σ^T is a block of every $f \in F$, then it is a block of every $f \in \text{closure}(F)$.

Strategy for Symmetric Reduction

1. Identify a set F of generator mappings over state-space of module such that
 - every $f \in F$ is an automorphism
 - the invariant is a block of every $f \in F$
2. Consider the stable partition \cong^F
3. Solve the problem on the quotient wrt \cong^F

Automorphisms for Pete

- f toggles x_1 and toggles x_2 : $t = f(s)$ iff

$$x_1[t] \neq x_1[s] \text{ and } x_2[t] \neq x_2[s],$$

$$pc_1[t] = pc_1[s] \text{ and } pc_2[t] = pc_2[s].$$
- g swaps the values of pc_1 and pc_2 , and if pc_2 is outC, toggles x_2 else toggles x_1 : $t = g(s)$ iff

$$pc_1[t] = pc_2[s] \text{ and } pc_2[t] = pc_1[s], \text{ and}$$
 if $pc_2[s] = outC$ then

$$x_1[t] = x_1[s] \text{ and } x_2[t] \neq x_2[s]$$
 else $x_1[t] \neq x_1[s]$ and $x_2[t] = x_2[s]$
- Subgroup generated by f and g : $\{f, g, id, f \circ g\}$
- Resulting partition has 12 classes of which 10 are reachable
- The region $\llbracket \neg(pc_1 = inC \wedge pc_2 = inC) \rrbracket$ is invariant under both functions f and g

Symmetry in Star Topology

- Server module P communicating with identical clients P_1, \dots, P_n
- Swapping controlled vars of two clients is an automorphism
- Set of generators: For every $1 \leq i, j \leq n$, an automorphism f_{ij}
- Partition \cong^F : one state is obtained from the other by arbitrary permutation of indices of client vars
- Example: Railroad controller

Symmetry in Ring Topology

- identical modules P_1, \dots, P_n connected in a ring
- Each module communicates with its neighbours
- Single generator: rotate left one step: $t = f(s)$
if $\text{ctr}X_{P_{i+1}}[t] = \text{ctr}X_{P_i}[s]$.
- The subgroup generated by f : rotate left arbitrarily many steps
- Example: dining philosophers, leader election, agreement

On-the-fly Reduction

- How do we exploit symmetries during on-the-fly search?
- Find a set of representative states, one per equivalence class of \cong^F
- Find a mapping rep that maps each state to its representative
- Explore only representatives:
 - Replace $InitQueue$ by $rep \circ InitQueue$
 - Replace $PostQueue$ by $rep \circ PostQueue$
- Orbit problem: given a set of generators (permutations of indices), determine if two states are equivalent.
- Orbit problem is as hard as graph isomorphism (no known polynomial solution)

Symmetry in Practice

- Enumerative model checker Mur φ at Stanford
- User-specified automorphisms in syntax:
 - Repetition type with restricted set of operations
 - Permuting indices in repetition type is guaranteed to be an automorphism
- On-the-fly symmetry reduction during DFS:
 - Each equivalence class can have multiple representatives
 - Mapping states to representatives is efficient