Motivation

- Previous lectures: How to determine satisfiability of propositional formulas
- Sometimes need to efficiently represent all solutions (i.e., satisfying assignments) to the formula
- Binary decision diagrams (BDDs): compact representation of all satisfying assignments of formula

Historical Context

- Invented by Randal Bryant from CMU; introduced in very influential 1986 paper
- BDDs have many applications: hardware and software verification, computer aided design of circuits, relational databases, ...
- Don Knuth: “One of the really fundamental data structures that came out in the last 25 years”

Binary Decision Trees

- Before talking about BDDs, let us first consider a simpler data structure called binary decision tree
- Binary decision tree: A tree data structure with two kinds of vertices, terminal and non-terminal
- Terminal vertices: boolean constants $\top$ (1) and $\bot$ (0)
- Non-terminal vertex labeled $v$ corresponds to boolean variable $v$ in propositional formula

Binary Decision Trees, cont.

- Each non-terminal vertex has two successors (i.e., edges)
- Low successor of non-terminal vertex $v$ (labeled with dashed edge) corresponds to assigning 0 to $v$
- High successor of vertex $v$ (labeled with solid edge) corresponds to assigning 1 to $v$
- Each path from root to a terminal node corresponds to an interpretation for the formula
- Paths ending in 1 correspond to satisfying interpretations
- Paths ending in 0 correspond to falsifying interpretations

Example: Binary Decision Tree

- Binary decision tree for formula with variables $x_1,x_2,x_3$:

  - Question: Is interpretation $x_1 = \bot$, $x_2 = \bot$, $x_3 = \top$ satisfying?
  - Question: What about $x_1 = \top$, $x_2 = \bot$, $x_3 = \top$?
  - This BDT is ordered: Any path from the root to a terminal contains variables in the same order (order: $x_1 < x_2 < x_3$)
Binary Decision Tree vs. Truth Table

- Binary decision tree encodes all satisfying assignments, but how does it compare to truth tables?
- Good news: Not as bad as it looks; there is a lot of redundancy! (e.g., different subparts are isomorphic)
- Idea: Merge redundant subparts and compress BDT into a much more space-efficient DAG representation!

Reduced Ordered Binary Decision Diagram

- To create compact representation, start with an ordered binary decision tree and apply a set of reduction rules
- The resulting data structure is called a reduced ordered binary decision diagram (ROBDD)
- When we talk about BDDs, we really mean ROBDDs (and so does everyone else)

Reduction Rule #1

Reduction rule #1: Merge all terminal nodes 1 into one node, and merge all terminal nodes 0 into one node.

Example: Merging Isomorphic Subtrees

- Which subgraphs are isomorphic?

Reduction Rule #2

Reduction rule #2: Merge isomorphic subgraphs

- Two subgraphs are isomorphic if:
  1. Their root represents the same variable
  2. The subgraphs rooted at their low successors are isomorphic
  3. The subgraphs rooted at their high successors are also isomorphic

Reduction Rule #3

- Reduction rule #3: Remove redundant nodes.
- Node v is redundant if its low and high successors are the same.
- Why?
- Eliminating redundant node v: Remove v from the graph, and redirect incoming edge to v to v’s children
Example: Removing Redundant Nodes

▶ Which nodes are redundant?

Exposing New Reductions

▶ Question: Can removing redundant nodes expose new isomorphic subtrees? (i.e., do we need to apply reduction 2 again after reduction 3?)

▶ Example: Are there any isomorphic subparts in this graph?

Exposing New Reductions, cont.

▶ Question: Can merging isomorphic subgraphs expose new redundant nodes? (i.e., do we need to apply reduction 3 again after reduction 2?)

▶ Example: Are there any redundant nodes in this graph?

Applying Reductions

▶ As examples illustrate, need to apply reduction rules until fixed point

▶ Resulting data structure after exhaustive application of reduction rules is a ROBDD.

▶ ROBDD is more space efficient compared to the binary decision tree because it eliminates redundancies.

Building ROBDDs

▶ Starting with BDT and applying reduction rule useful way to understand BDD invariants

▶ But no one builds BDDs this way. Why?

▶ Idea: Build the ROBDD for a formula directly without building the binary decision tree!

Building BDDs directly from Formulas

▶ Consider a formula $\phi$ of the form $\phi_1 \star \phi_2$ where $\star$ is any boolean connective

▶ To construct the ROBDD for $\phi$, first recursively construct the ROBDDs for $\phi_1$ and $\phi_2$

▶ Then, combine BDDs for $\phi_1$ and $\phi_2$ to form the BDD for $\phi$

▶ To combine BDDs for $\phi_1$ and $\phi_2$, will use a technique called Shannon’s decomposition
Cofactoring and Shannon’s Decomposition

- Shannon’s decomposition involves an operation called cofactoring.
- Cofactoring a boolean formula restricts the formula to a particular value of a variable.
- Example: What is the resulting formula when we restrict \( x_1 \) to \( \top \) in \( x_1 \land x_2 \)?
- The positive cofactor \( \phi \downarrow x_i \) is the resulting formula when \( x_i \) is replaced by \( \top \).
- The negative cofactor \( \phi \downarrow \neg x_i \) is \( \phi \) with \( x_i \) replaced by \( \bot \).
- Example: What is \((x_1 \lor (\neg x_2 \land x_3)) \downarrow \neg x_2\)?

Cofactoring Using BDDs

- If we have a BDD for \( \phi \), it is easy to build BDD for positive and negative cofactors of \( \phi \) with respect to \( x \).
- Given BDD for \( \phi \), how do we build BDD for \( \phi \downarrow x \)?
- How do we build BDD for \( \phi \downarrow \neg x \)?

Shannon’s Decomposition

- Can now define Shannon’s decomposition using cofactors.
- \( \phi \) ≡ \((x \land \phi \downarrow x) \lor (\neg x \land \phi \downarrow \neg x)\)
- Basically a case analysis on \( x \)’s truth value
- If \( x \) is \( \top \), then the positive cofactor must be true
- If \( x \) is \( \bot \), then the negative cofactor must be true

Using Shannon’s Decomposition to Build BDD

- Now, we can directly build BDD for a formula \( \phi_1 \land \phi_2 \) using Shannon’s decomposition
- First, build BDD for subformulas \( \phi_1 \) and \( \phi_2 \)
- Then, use recursive procedure called Apply to build BDD for \( \phi \) from BDDs for \( \phi_1 \) and \( \phi_2 \)

The Base Case

- Suppose that the BDDs for \( \phi_1 \) and \( \phi_2 \) have root nodes \( x \) and \( x' \) and both are boolean constants
- Then, what is the BDD for \( \phi_1 \land \phi_2 \)?
Recursive Step

- Given root nodes \( x, x' \), suppose \( x \) precedes \( x' \) according to variable order.
- First do a case analysis on \( x \) using Shannon’s decomposition.
- Using Shannon’s decomposition, what is \( \phi_1 \ast \phi_2 \) equivalent to?
- Cofactoring distributes over connectives, so rewrite as:
  \[
  (x \land (\phi_1 \downarrow x \ast \phi_2 \downarrow x) \lor (\neg x \land (\phi_1 \downarrow \neg x \ast \phi_2 \downarrow \neg x))
  \]

Recursive step, continued

- Thus, we need to build BDD for:
  \[
  (x \land (\phi_1 \downarrow x \ast \phi_2 \downarrow x) \lor (\neg x \land (\phi_1 \downarrow \neg x \ast \phi_2 \downarrow \neg x))
  \]
- We know how to build BDDs for \( \phi_1 \downarrow x \) and \( \phi_1 \downarrow \neg x \).
- How do we build BDDs for \( \phi_1 \downarrow x \ast \phi_2 \downarrow x \) and \( \phi_1 \downarrow \neg x \ast \phi_2 \downarrow \neg x \)?

An Example

- Consider formula \( \phi_1 \lor \phi_2 \) where \( \phi_1 \) is \( x_1 \leftrightarrow x_2 \), and \( \phi_2 \) is \( \neg x_2 \).
- Suppose \( x_1 \) precedes \( x_2 \) in variable order.
- Assume we have BDDs for \( \phi_1 \) and \( \phi_2 \):

Example, continued

- Since \( x_1 \) precedes \( x_2 \) in variable order, what do we perform case split on?
- How do we decompose \( \phi_1 \lor \phi_2 \) using Shannon’s decomposition?
- Thus, we recursively compute BDDs for \( (\phi_1 \downarrow \neg x_1 \lor \phi_2 \downarrow \neg x_1) \) and \( (\phi_1 \downarrow x_1 \lor \phi_2 \downarrow x_1) \).
Example, continued

Final BDD:

![Final BDD Diagram]

Example 2

Give the BDD representing the negation of the following BDD:

![Example 2 BDD Diagram]

Nice BDD Properties: Canonicity

- BDDs are a **canonical** representation of boolean formulas.
- Canonical means two equivalent formulas have same representation.
- Thus, if we construct BDDs for two equivalent formulas using same variable order, resulting BDDs are the same.
- Are any of the normal forms we talked about (NNF, CNF, DNF) canonical?
- Example:

Consequences of Canonicity

- **Corollary 1**: Given BDDs for $\phi_1$ and $\phi_2$ constructed with same variable order, equivalence of $\phi_1$ and $\phi_2$ just syntactic check.
- **Corollary 2**: Given BDD for formula $\phi$, $\phi$ is unsatisfiable if and only if its BDD representation is the boolean constant 0.
- How does this corollary follow from canonicity?
- **Corollary 3**: Given BDD for formula $\phi$, $\phi$ is valid if and only if its BDD representation is the boolean constant 1.

BDD Size and Variable Order

- Another important BDD property: Size of a BDD for a given formula $\phi$ is very sensitive to variable order.
- For some variable orders, the size of the BDD may be only **polynomial** in the number of variables.
- For some other variable orders, the size of the BDD for same formula may be **exponential**.
- Furthermore, there are boolean formulas for which any variable order causes an exponential blow-up.

Choosing Variable Order

- Since size of BDD is so sensitive to variable order, we would like to construct BDD using a good variable order.
- Unfortunately, NP-complete to decide whether a given order is optimal.
- Typically, heuristics are used to predict good variable order.
- Heuristics for finding good variable order can be either **static** (i.e., determined up-front) or **dynamic** (i.e., change as BDD operations proceed).
- Dynamic orders typically yield more compact BDDs, but slower.
BDDs can be used to compactly represent all satisfying assignments to a boolean formula

Pros:

+ Canonical representation ⇒ checking equivalence, validity, satisfiability constant time operations once BDD is built
+ If we have good variable order, can yield a compact representation of all satisfying assignments of formula

Cons:

- Compactness very sensitive to variable order
- Can cause an exponential blow-up in the representation of the formula