Propositional Logics
Formal Methods
Lecture 2

Farn Wang
Dept. of Electrical Engineering
National Taiwan University

Outline

- What is verification?
- What is logic?
- Propositional Logic
- BDD for Propositional Logic

What is Verification?

Verification involves checking a satisfaction relation, usually in the form of a sequent :

$$M \models \varphi$$

Where

- M is a model (or implementation)
- φ is a property (or specification)
- ⊨ is a relationship that should hold between M and φ, i.e., $(M,\varphi) \in \models$

What is Verification?

- Verification involves:
  - specifying the model / system / implementation
  - specifying the property / specification
  - choosing the satisfaction relation
  - checking the satisfaction relation

These 4 steps are NOT independent.

Example: specify the model as a finite state machine, specify the property in temporal logic, use the satisfaction relation that the model must satisfy a formula in temporal logic, use model checking to check the satisfaction relation.

Logic and Verification

- Different logics
  - give us different ways of expressing M and φ and
  - define the pairs that are members of ⊨.
- Another way to say this is to say
  - that the model satisfies the property, or
  - that we can conclude the property from the model.
- Hopefully the calculation of the satisfaction relation is compositional in either the property or the model.
  - This decomposes the verification task.
- The model and property both describes sets of "behaviours".
- The satisfaction relation is a relation between the set of behaviours of the model and the set of behaviours of the property.

Models and Properties

- The term "model" is used loosely here. It may not be executable, and it may not be a complete description of the system's behaviour.
  - The terms implementation and specification are relative terms.
  - An implementation generally contains more details than a specification.
- In hardware, often the model is a description of the circuit in a hardware description language such as VHDL or Verilog.
  - The real thing is the physical realization of the chip.
- Sometimes the model is actually a specification and the property is an attribute such as completeness or consistency.

What is a Logic?

- In general, logic is about reasoning.
- It is about the validity of arguments, consistency among statements (. . .) and matters of truth and falsehood.
- In a formal sense logic is concerned only with the form of arguments and the principles of valid inferencing.

Another definition of Logic

- logic is: the science of correct reasoning, valid induction or deduction.
- Symbolic logic is a modern type of formal logic using special mathematical symbols for
  - propositions,
  - quantifiers, and
  - relationships

  among propositions and concerned with the elucidation of permissible operations upon such symbols.

Induction vs Deduction

- These are two branches in the philosophical study of logic.
- Induction is “the process of deriving general principles from particular facts or instances. ”
- Example:
  - Coffee shop burger #1 was greasy.
  - Coffee shop burger #2 was greasy.
  - Coffee shop burger #3 was greasy....
  - Coffee shop burger #100 was greasy.
  - Therefore, all coffee shop burgers are greasy.
- In induction, conclusions are probable but not conclusive.

Induction vs Deduction

- Deduction is “the process of reasoning in which a conclusion follows necessarily from the stated premises;
  - inference by reasoning from the general to the specific. ”
- Mathematical Induction: a method of proving statements about well-ordered sets.
  - The most common use of mathematical induction is for the natural numbers where there is a base case and an induction hypothesis.
  - Mathematical induction is a form of deduction because the conclusions are conclusive.
- We will be studying deduction and using mathematical induction .

Another definition of Logic

- A branch of philosophy and mathematics that deals with the formal principles, methods and criteria of validity of inference, reasoning and knowledge.
- Logic is concerned with what is true and how we can know whether something is true.
  - This involves the formalization of logical arguments and proofs in terms of symbols representing propositions and logical connectives.
  - The meanings of these logical connectives are expressed by a set of rules which are assumed to be self-evident .
- In symbolic logic, arguments and proofs are made in terms of symbols representing propositions and logical connectives.
  - The meanings of these begin with a set of rules or primitives which are assumed to be self-evident.
  - Fortunately, even from vague primitives, functions can be defined with precise meaning.

Elements of a Logic

- A logic consists of:
  - syntax
  - semantics
  - proof procedure(s) (also called proof theory)

Syntax and Semantics

- syntax:
  - define “well-formed formula”
- semantics:
  - define “$\models$” (“satisfies”)

    M $\models \phi$ (satisfaction relation)
  - define $\phi_1, \phi_2, \phi_3 \models \psi$ (“entails”, or semantic entailment) means from the *premises* ϕ_1, ϕ_2, ϕ_3, we may conclude ψ,
    - ϕ_1, ϕ_2, ϕ_3, and ψ are all well-formed formulae in the logic

Proof Procedure

- define “ $\vdash$” (pronounced “proves”)
- a proof procedure is a way to calculate $\phi_1, \phi_2, \phi_3 \vdash \psi$ (also called a sequent).
  - By “calculation”, we mean that there is a procedure for determining if

    $$(\phi_1, \phi_2, \phi_3, \psi) \in \vdash$$
- there may be multiple proof procedures that we will indicate by subscripting $\vdash$,
  - e.g., the natural deduction proof procedure for propositional logic will be $\vdash_{ND}$
- for some logics, there isn’t a proof procedure that always terminates for any sequent

Proof Procedures

- Proof procedures are algorithms that perform “mechanical manipulations on strings of symbols.
  - A proof procedure does not make use of the meanings of sentences,
  - it only manipulates them as formal strings of symbols”.
- There may be multiple ways to prove a sequent in a particular proof procedure.
- A theory is the set of theorems that can be proven by a proof procedure.

Theorem Provers

- Many proof procedures rely on pattern matching, i.e., looking for statements that have the same form with appropriate substitutions (unification).
- As we work with multiple proof procedures, we will see that working through the steps is often very mechanical, i.e., the kinds of things that computers do well!
- Theorem provers are software tools that mechanize proof procedures.
  - Theorem provers can be interactive or automatic.

Soundness and Completeness

- The semantics and the proof procedures ($\models$ and $\vdash$) are related in the concepts of soundness and completeness.

  Definition. *A proof procedure is sound if* $\phi_1, \phi_2, \phi_3 \vdash \psi$ *then* $\phi_1, \phi_2, \phi_3 \models \psi$*.*

  A proof procedure is sound if it proves only tautologies.

  Definition. *A proof procedure is complete if* $\phi_1, \phi_2, \phi_3 \models \psi$ *then* $\phi_1, \phi_2, \phi_3 \vdash \psi$*.*
- A proof procedure is complete if it proves every tautology.
- Note that in the literature, there is not consistent use of the symbols $\models$ and $\vdash$.

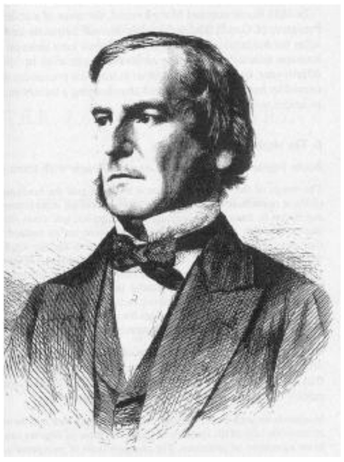
Consistency

- **Definition.** *A proof procedure is consistent if it is not possible to prove both* $\mathcal{A}$ *and* $\neg\mathcal{A}$*, i.e.,*

  not both $\vdash \mathcal{A}$ *and* $\vdash \neg\mathcal{A}$ *.*

Propositional Logic

- Invented by George Boole (1815-64). “An Investigation of the Laws of Thought on which are founded The Mathematical Theories of Logic and Probabilities”.



Propositional Logic

Propositional logic is also called *sentential* logic, i.e., the logic of sentences. It is also called propositional calculus or sentential calculus.

- syntax (well-formed formulas)
- semantics (truth tables)
- proof theory
  - axiom systems
  - natural deduction
  - sequent calculus

Propositional Logic: Syntax

Its syntax consists of:

- two constant symbols: **true** and **false**
- Proposition letters
- Propositional connectives
- Brackets

Propositions

Definition. *Proposition letters represent declarative sentences, i.e., sentences that are true or false.*

Sentences matching proposition letters are atomic (non-decomposable), meaning they don't contain any of the propositional connectives.

Examples:

- It is raining outside.
- The sum of 2 and 5 equals 7.
- The value of program variable *a* is 42.

Sentences that are interrogative (questions), or Imperative (commands) are not propositions.

Using Symbols

- Because in logic, we are only concerned with
  - the structure of the argument and
  - which structures of arguments are valid,

  we "encode" the sentences in symbols to create a more compact and clearer representation of the argument.
- We call these propositional symbols or proposition letters.
- DO NOT use *T, F, t,* or *f* in any font as symbols representing sentences!

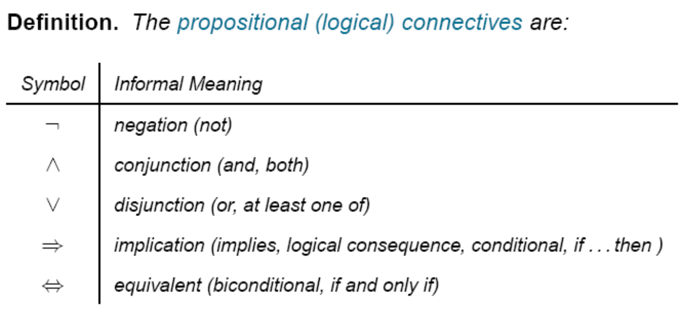
Example of Using Symbols

- If the train arrives late and there are no taxis at the station, then John is late for his meeting.
- John is not late for his meeting.
- The train did arrive late.
- Therefore, there were taxis at the station.

$$\begin{array}{l} (p \wedge \neg q) \Rightarrow r \\ \neg r \\ p \\ \hline q \end{array}$$

Prop. Letter	Sentence
p	the train is late
q	there are taxis at the station
r	John is late for his meeting

Propositional Connectives



Definition. *The propositional (logical) connectives are:*

Symbol	*Informal Meaning*
$\neg$	*negation (not)*
$\wedge$	*conjunction (and, both)*
$\vee$	*disjunction (or, at least one of)*
$\Rightarrow$	*implication (implies, logical consequence, conditional, if . . . then)*
$\Leftrightarrow$	*equivalent (biconditional, if and only if)*

Others may use different symbols for these operations.

Terminology

In implication, as in $p \Rightarrow q$

- p is the premise or antecedent or hypothesis
- q is the consequent or conclusion

$\neg b \Rightarrow \neg a$ is called the contrapositive of $a \Rightarrow b$.

The set of connectives $\{\neg, \wedge\}$ are complete in the sense that all the other connectives can be defined using them, e.g., $a \vee b = \neg(\neg a \wedge \neg b)$.

Other subsets of the binary connectives are also complete in the same sense.

Well-formed formulas

- The following is an expression formed out of propositional symbols, brackets propositional connectives:

$$a(\wedge c \Rightarrow)b$$

- but it’s not a formula in propositional logic!

- Next, we make precise the notion of a well-formed formula

Well-formed formulas

Definition. *The well-formed formulae of propositional logic are those obtained by the following construction rules:*

- **true**, **false**, *and the proposition letters are atomic formulas.*
- *If* a *is an atomic formula, then* a *is a formula.*
- *If* p *and* q *are formulas, then each of the following are formulas:*

$$(\neg p)\ (p \wedge q)\ (p \vee q)\ (p \Rightarrow q)\ (p \Leftrightarrow q)$$

No other expressions are formulas.

Note that this is an inductive definition, meaning the set is defined by basis elements, and rules to construct elements from elements in the set.

Well-formed Formulas

- Brackets around the outermost formula are usually omitted.
- Brackets can be omitted using the following rules of precedence of operators:

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

Note: Some texts do not use exactly these rules of precedence, they rank $\wedge$ and $\vee$ at the same level of precedence, and $\Rightarrow$ and $\Leftrightarrow$, at the same level of precedence.

29

Semantics

- Semantics means “meaning”.
- Semantics relate two worlds. Semantics provide an interpretation (mapping) of expressions in one world in terms of values in another world. Semantics are often a function from expressions in one world to expressions in another world.
- The semantics (i.e., the mapping) is often called a model or an interpretation. We write $\mathcal{M} \vDash \phi$ to mean the model satisfies the formula. In propositional logic, models are called Boolean valuations.
- Proof procedures transform the syntax of a logic in ways that respect the semantics.

30

Semantics of Propositional Logic

- We've described the syntax for propositional logic, which is the domain of the semantic function.
- Classical logic is two-valued. The two possible truth values are T, and F, which are two distinct values.
- The range of the semantic function for propositional logic is the set of truth values:

$$\mathrm{Tr} = \{\mathrm{T},\mathrm{F}\}$$

- Note that these truth values are distinct from the syntax elements **true**, and **false**.

Semantics of Propositional Logic

- Truth Values: $\mathrm{Tr} = \{\mathrm{T},\mathrm{F}\}$
- There are functions on these truth values that correspond to the meaning of the propositional connectives. We overload the operators " $\wedge$ ", " $\vee$ ", etc. to be both part of the syntax of propositional logic, and operations on the sets of truth values in our model for propositional logic.

$$\neg : \mathrm{Tr} \rightarrow \mathrm{Tr}$$
$$\wedge : (\mathrm{Tr} \times \mathrm{Tr}) \rightarrow \mathrm{Tr}$$
$$etc.$$

- Truth tables are used to describe the functions of operations on these truth values.

Truth Tables

p	$\neg p$
T	F
F	T

p	q	$p \wedge q$	$p \vee q$	$p \Rightarrow q$	$p \Leftrightarrow q$
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

- These connectives are truth functional, that is given the truth values "of each component part of a compound sentence containing connectives, the truth value of the whole sentence is uniquely determined".
- A truth table for any formula containing n atomic propositions has 2^n lines.

Boolean Valuations

- The semantics of propositional logic are an interpretation of any expression in propositional logic (i.e., the constants **true** and **false**, the proposition letters, and the propositional connectives) in the set Tr. The semantics of propositional logic are called a Boolean valuation.
- **Definition.** *A Boolean valuation is a mapping v from the set of propositional formulas to the set Tr meeting the conditions:*
  - $v(\text{true})=T$, $v(\text{false})=F$.
  - $v(\neg p) = \neg(v(p))$.
  - *for all the connectives:* $v(p \circ q) = v(p) \circ v(q)$
- Note that : $\neg(v(p))$ and $v(p) \circ v(q)$ is given by the truth tables on the previous slide.

Boolean Valuations

- Here's an example of a Boolean valuation:
  $$v(p)=\mathrm{T},\ v(q)=\mathrm{F},\ v(r)=\mathrm{F},\ v(\text{false})=\mathrm{F},\ v(\text{true})=\mathrm{T}$$
  and the propositional connectives map to the corresponding operation on the truth values in the model.
  $$\begin{aligned} v((p\Rightarrow q)\wedge r) &= v(p\Rightarrow q)\wedge v(r) \\ &= (v(p)\Rightarrow v(q))\wedge v(r) \\ &= (\mathrm{T}\Rightarrow \mathrm{F})\wedge \mathrm{F} \\ &= \mathrm{F}\wedge \mathrm{F} \\ &= \mathrm{F} \end{aligned}$$
- A Boolean valuation is uniquely determined by the values of v for the proposition letters. There are multiple Boolean valuations for propositional logic.

Satisfiability

- **Definition.** *A formula* a *is* *satisfiable* *if there is a Boolean valuation* v *such that* $v(a)=\mathrm{T}$
- We sometimes say that the formula "has a satisfying assignment" to mean that it is satisfiable.
- We are mostly interested in the propositional formulas that map to T in all the possible Boolean valuations (i.e., in all model).

Tautologies

- **Definition.** *A propositional formula* a *is a tautology (also called valid or a theorem) if* $v(a) = \mathrm{T}$ *for every Boolean valuation* v*.*

  i.e., , a tautology is a formula that is true for all possible truth values of the propositional letters used in the formula. The last column of the truth table for a tautology contains all T.
- Note that a formula a is a tautology iff : $\neg a$ is not satisfiable.

37

Semantic Entailment

$$\phi_1, \phi_2, \phi_3 \vDash \psi$$

means that if $v(\phi_1) = \mathrm{T}$ and $v(\phi_2) = \mathrm{T}$ and $v(\phi_3) = \mathrm{T}$ then $v(\psi) = \mathrm{T}$, which is equivalent to saying

$$(\phi_1 \wedge \phi_2 \wedge \phi_3) \Rightarrow \psi$$

is a tautology, i.e.,

$$(\phi_1, \phi_2, \phi_3 \vDash \psi) \equiv ((\phi_1 \wedge \phi_2 \wedge \phi_3) \Rightarrow \psi)$$

38

Models and Entailment

- In propositional (and predicate) logic, $\models$ is overloaded and has two meanings:
- $\mathcal{M} \models \phi$ relates a model to a formula, saying that $\mathcal{M}$ satisfies the formula ϕ. This is called a *satisfaction relation*.
- $\psi \models \phi$ relates two formulas, saying that for all v (i.e., for all possible models), if $v(\psi) = \mathrm{T}$ then $v(\phi) = \mathrm{T}$. This is called *semantic entailment*.
- These two uses can be distinguished by their context.

Falsehood

Definition. *A falsehood (contradiction) is a formula that is false for all possible truth values of the propositional symbols used in the formula.*

- *The last column of the truth table for a contradiction contains all F.*

Consistency

Definition. *A collection of formulas is consistent if the formulas can all be true simultaneously.*

- A collection of formulas is consistent if there is a Boolean valuation in which all the formulas can be true simultaneously.

Consistency

- If a set of premises of an implication are not consistent, they can be used to prove a contradiction, i.e.,

$$p, \neg p \models q \wedge \neg q$$

or

$$p, \neg p \models \text{false}$$

- This is sometimes called the “false implies anything” problem, meaning that nothing is proven about a system if there are inconsistent premises.
- It is standard practise in verification to check that one’s premises are not inconsistent to avoid this problem.

Example of Checking Consistency

- Sales of houses fall off if interest rates rise. Auctioneers are not happy if sales of houses fall off. Interest rates are rising. Auctioneers are happy.
  - s = sales of houses fall off
  - r = interest rates rise
  - h = auctioneers are happy
  - The formulas of the problem are: $r \Rightarrow s, s \Rightarrow \neg h, r, h$
- To check that this set of formulas is consistent, we check that
  - the conjunction of the formulas is satisfiable
  - i.e., there is a Boolean valuation that maps the formula to T,
  - i.e., that the conjunction of the formulas is not a contradiction.

Example of Checking Consistency

Does the following have a satisfying assignment?

$$(r \Rightarrow s) \wedge (s \Rightarrow \neg h) \wedge r \wedge h$$

Example of Checking Consistency

- Thus, the conjunction of the formulas is a contradiction so this set of formulas is inconsistent.
- Using the proof procedures that we will talk about next, to prove a set of formulas is inconsistent, we would prove that the negation of the conjunction of the formulas is a tautology.

Decidability

- A question is decidable if there is an algorithm that will always terminate and deliver the correct answer to the problem "yes" or "no".
- A logic is decidable if there is an algorithm to determine if any formula of the logic is a tautology (is a theorem, is valid).
- Propositional logic is decidable because we can always construct the truth table for the formula.

Proof Procedures

- We can always determine if a formula is a tautology
  - by using truth tables to determine the value of the formula for every possible combination of values for its proposition letters,
  - but this would be very tedious since the size of the truth table grows exponentially .
- Proof procedures for propositional logic are alternate means to determine tautologies.
  - As long as the proof procedure is sound, we can use the proof procedure in place of truth tables to determine tautologies.

Proof Procedures for Propositional Logic

- There are many proof procedures for propositional logic.
- Some match the human reasoning process.
- Others are better suited to automation by computers.

Examples of proof procedures are:

- Resolution
- Semantic Tableaux
- Natural Deduction
- Sequent Calculus
- Hilbert Systems (axiom systems)
- Davis-Putnam
- Binary Decision Diagrams

Proof Procedures for Propositional Logic

- ". . . based on different insights into the processes by which one recognizes that a formula expresses a logical truth."
- As an appropriate lead-in to interactive theorem provers, we will begin by studying two procedures that match human reasoning and are related to the way proofs are conducted in a theorem prover:
  - natural deduction and
  - sequent calculus.

Proof Styles

- A proof procedure is a set of rules we use to transform premises and conclusions into new premises and conclusions.
- A goal is a formula that we want to proof is a tautology. It has premises and conclusions.
- A proof is a sequence of proof rules that when chained together relate the premise of the goal to the conclusion of the goal.

Forward and Backward Proof

- In forward proof, we work from premises to conclusions.
  - We apply rules that infer new formulas from premises.
  - After many steps, the final infered formulas should match the conclusion to have a proof.
- In backward proof, we work from conclusions to premises.
  - We use the proof rules backwards to reduce a conclusion to a formula closer to the premises.
  - After many steps, the final reduced formula should match the premise.
- Forward proofs are easy to explain, but hard to find.

Hilbert Systems

- Also called axiom systems or Frege systems.
  - Axiom systems are forward reasoning.
  - Starting with known tautologies,
  - derive immediate consequences,
  - continue this until the desired formula is reached.
- In axiom systems, we use axioms and rules of inference (also called rules of derivation).
- The following discussion is general for all Hilbert systems,
  - not just those for propositional logic.

Derivations

Definition. *A derivation in a Hilbert system from a set S of formulas is a finite sequence $X_1, X_2, \ldots, X_n$ of formulas such that each term*

- *is either an axiom,*
- *or is a member of S,*
- *or follows from earlier terms by one of the rules of inference.*

We write:

$$S \underset{ph}{\vdash} X$$

to say that X has a derivation from S in the propositional Hilbert system.

53

Proofs

Definition. *A proof in a Hilbert system is a finite sequence*

$$X_1, X_2, \ldots, X_n$$

of formulas such that each term is either an axiom or follows from earlier terms by one of the rules of inference.

- *A proof is a derivation from an empty set of formulas, i.e.,* $\underset{ph}{\vdash} X$

  We will write proofs
  - as a list of formulas,
  - each on its own line, and
  - refer to the line of a proof in the justification for steps.

Definition. X *is a theorem of a Hilbert system if* X *is the last line of a proof.*

X *is a consequence of a set* S *if* X *is the last line of a derivation from* S*.*

54

Hilbert System for Propositional Logic

- Every axiom must be a tautology.
  - Rules of inference produce tautologies from tautologies.
- It's *not* very interesting (or useful) to take all the tautologies as axioms,
  - rather we need a finite number of axioms,
  - or at least a finite number of forms that axioms can take.
  - We call these forms axiom schemes.
- For example, all $p \Rightarrow p, (p \wedge q) \Rightarrow (p \wedge q)$ and $\neg q \Rightarrow \neg q$ have the form $X \Rightarrow X$.
- We adopt the convention of using capital letters to represent formulas in axiom schemes.

AL – *an Axiomatic System for Prop. Logic*

- We limit ourselves to two connectives $\neg$, and $\Rightarrow$, and rewrite any expressions involving other connectives in terms of these two.
  - Note that this is a complete set of operators.
- Three axiom (schemes):
  - $A \Rightarrow (B \Rightarrow A)$.
  - $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$.
  - $(\neg A \Rightarrow \neg B) \Rightarrow (B \Rightarrow A)$.
- One rule of inference:
  - (*modus ponens* - MP) From A and $A \Rightarrow B$, B can be derived, where A and B are any well-formed formulas.

Simple Example of a Proof

- Show $\underset{ph}{\vdash}((x \Rightarrow y) \Rightarrow (x \Rightarrow x))$:
  - $x \Rightarrow (y \Rightarrow x)$.

    Ax1 where $A \square\, x, B \square\, y$
  - $(x \Rightarrow (y \Rightarrow x)) \Rightarrow ((x \Rightarrow y) \Rightarrow (x \Rightarrow x))$.

    Ax2 where $A \square\, x, B \square\, y, C \square\, x$
  - $(x \Rightarrow y) \Rightarrow (x \Rightarrow x)$.

    MP on lines 1 and 2

Example

- Rather than constructing particular proofs, we can actually construct "meta-theorems" (theorem schemes).
- Example: Show $\underset{ph}{\vdash} A \Rightarrow A$

Examples to Try

- Show the following:
  - $\vdash_{ph} \neg A \Rightarrow (A \Rightarrow B)$.
  - $\{A \Rightarrow B, B \Rightarrow C\} \vdash_{ph} A \Rightarrow C$.
  - $\vdash_{ph} (B \Rightarrow C) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$.

Note: You can reuse previous results in these proofs.

Deduction Theorem

- **Theorem.** *In any Hilbert System with at least Axiom Schemes 1 and 2, and with Modus Ponens as the only rule of inference,*

$$S \cup \{X\} \vdash_{ph} Y \text{ iff } S \vdash_{ph} (X \Rightarrow Y)$$

- This result was proven by both Tarski and Herbrand.

Use of the Deduction Theorem

- Show $\{A \Rightarrow B\} \vdash_{ph} A \Rightarrow (C \Rightarrow B)$
- Set out to show: $A \Rightarrow B, A \vdash_{ph} (C \Rightarrow B)$:

A	premise
$A \Rightarrow B$	premise
B	MP on 1 and 2
$B \Rightarrow (C \Rightarrow B)$	Ax1
$C \Rightarrow B$	MP on 3 and 4

- Now that we've proven $\{A \Rightarrow B, A\} \vdash_{ph} C \Rightarrow B$, using the deduction theorem we can conclude:

$$\{A \Rightarrow B\} \vdash_{ph} A \Rightarrow (C \Rightarrow B)$$

Soundness and Completeness of AL

- (Soundness) Every theorem A in AL is a tautology: $\vdash_{ph} A \quad \Rightarrow \quad \vDash A$
- (Completeness) If A is a tautology then it is a theorem of AL: $\vDash A \quad \Rightarrow \quad \vdash_{ph} A$
- AL is consistent.

An Aside on Monotonicity

- **Definition.** *A* *monotonic logic* *is one where a valid proof cannot be invalidated by the addition of extra premises.*
- We will only be studying monotonic logics.
- Non-monotonic logics are often useful for reasoning about knowledge.

63

2007/03/13 stops here.

64

Proof Procedure: Natural Deduction

- Natural deduction is a collection of proof rules, each of which allows us to infer formulas from other formulas, eventually to get from a set of premises to a conclusion.
- Natural deduction is a form of forward proof.
  - Starting from the premises, we use the inference rules to deduce new formulas that logically follow from the premises.
  - We continue this process until we have deduced the conclusion.

Natural Deduction

$$p_1, p_2, p_3, \ldots \vdash_{ND} q$$

- The notation above means that there is a proof using natural deduction that the argument with premises $p_1, p_2, p_3, \ldots$ and conclusion q is valid.
- Logical formulas ψ such that $\vdash_{ND} \psi$ are called theorems.
- Again, there are multiple natural deduction systems for propositional logic.
  - We will be following the presentation of Huth and Ryan.

Natural Deduction

- Gerhard Gentzen (1909–1945). Natural deduction was introduced in his paper *Investigations into Logical Deduction*, 1935.

67

Inference Rules

Definition. *An inference rule is a primitive valid argument form.*

- *Each inference rule enables the elimination or the introduction of a logical connective.*
- Most inference rules have names that consists of:
  - a logical connective,
  - a letter: $\wedge$i, $\Rightarrow$e
    - "i" indicates that the rule introduces the connective
    - "e" indicates that the rule eliminates the connective

68

Natural Deduction

- Natural deduction is based on the idea of subordinate proofs.
  - We make assumptions, and then discharge the assumptions.
- Subordinate proofs are indented/boxed with the first line in the box being the assumption made in that subordinate proof.
  - The first line below the indentation/box is the result of discharging the assumption.
- The formulas active at a stage in the proof are those occurring in boxes that haven't yet been closed.
  - We can only use active formulas to derive new formulas.
- The rules come in pairs: one for introducing a connective and one for eliminating it.

Rules for Conjunction

and-introduction

$$\frac{p \quad q}{p \wedge q}\wedge\mathrm{i}$$

and-elimination

$$\frac{p \wedge q}{p}\wedge\mathrm{e}_1$$

$$\frac{p \wedge q}{q}\wedge\mathrm{e}_2$$

- Above the line are the premises of the rule.
- Below the line is the conclusion.
- To the right of the line is the name of the rule.
- p and q may be larger formulas than proposition letters.
- It's okay to just use $\wedge\mathrm{e}$ and not distinguish $\wedge\mathrm{e}_1$ from $\wedge\mathrm{e}_2$

Example #1

- Show $p \wedge q, r \vdash_{ND} q \wedge r$
  - $p \wedge q$ premise
  - r premise
  - q $\wedge$e 1
  - $q \wedge r$ $\wedge$i 2,3
- We present proofs in the linear format, but a tree format could be used.
- Try: Show $(p \wedge q) \wedge r, s \wedge t \vdash_{ND} q \wedge s$

Rules for Double Negation

$$\frac{\neg\neg p}{p}\,\neg\neg\mathrm{e} \qquad\qquad \frac{p}{\neg\neg p}\,\neg\neg\mathrm{i}$$

Example #2

- Show $p, \neg\neg(q \wedge r) \vdash_{ND} \neg\neg p \wedge r$

Rules for Eliminating Implication

- Implies-elimination $$\frac{p \quad p \Rightarrow q}{q} \Rightarrow e$$

  This is modus ponens.
- We can also derive modus tollens:
$$\frac{p \Rightarrow q \quad \neg q}{\neg p}\mathrm{MT}$$
- Example: If it is raining, then I have my umbrella up. I do not have my umbrella up. Therefore it is not raining.

Example #3

- Show $\neg p \Rightarrow q, \neg q \vdash_{ND} p$

Implies Introduction

$$\frac{\left[\begin{array}{l} r \quad \text{assumption} \\ \vdots \\ q \end{array}\right.}{r \implies q} \Rightarrow \mathrm{i}$$

Within the box, we assume r, and then prove q. The box marks the scope of the temporary assumption. Any lines in the box depend on the assumption. The line after the box discharges the assumption by moving it to the LHS of the implication on the RHS. The line after the box no longer depends on the assumption. Boxes may be nested.

We can only use a formula in the proof if it occurs prior to this line in the proof and it doesn't occur within an enclosed box (i.e., it is active). We can copy a formula that has appeared before as long as it is still active.

Example #4

- Show $\vdash_{ND} p \Rightarrow p$

[1	p	assumption
2	$p \Rightarrow p$	$\Rightarrow$ i $1-1$

Example #5

- Show $\vdash_{ND} (q \Rightarrow r) \Rightarrow ((\neg q \Rightarrow \neg p) \Rightarrow (p \Rightarrow r))$

1	$q \Rightarrow r$	assumption
2	$\neg q \Rightarrow \neg p$	assumption
3	p	assumption
4	$\neg\neg p$	$\neg\neg$i 3
5	$\neg\neg q$	MT $2, 4$
6	q	$\neg\neg$e 5
7	r	$\Rightarrow$ e $1, 6$
8	$p \Rightarrow r$	$\Rightarrow$ i $3-7$
9	$(\neg q \Rightarrow \neg p) \Rightarrow (p \Rightarrow r)$	$\Rightarrow$ i $2-8$
10	$(q \Rightarrow r) \Rightarrow ((\neg q \Rightarrow \neg p) \Rightarrow (p \Rightarrow r))$	$\Rightarrow$ i $1-9$

Examples to Try

- Show
  - $p \wedge q \Rightarrow r \vdash_{ND} p \Rightarrow (q \Rightarrow r)$
  - $p \wedge (q \Rightarrow r) \vdash_{ND} p \wedge q \Rightarrow r$
  - $p \Rightarrow q \vdash_{ND} (p \wedge r) \vdash_{ND} (q \wedge r)$

Rules for Disjunction

or-introduction

$$\frac{p}{p \vee q}\vee\mathrm{i}_1 \qquad \frac{q}{p \vee q}\vee\mathrm{i}_2$$

or-elimination
(case analysis)

$$\frac{p \vee r \qquad \boxed{\begin{matrix} p \\ \vdots \\ q \end{matrix}} \qquad \boxed{\begin{matrix} r \\ \vdots \\ q \end{matrix}}}{q}\vee\mathrm{e}$$

It's okay to just use $\vee\mathrm{i}$ and not distinguish $\vee\mathrm{i}_1$ from $\vee\mathrm{i}_2$.

Example # 6

- Show $p \vee q \vdash_{ND} q \vee p$

1	$p \vee q$	premise
2	p	assumption
3	$q \vee p$	$\vee$i 2
4	q	assumption
5	$q \vee p$	$\vee$i 4
6	$q \vee p$	$\vee$e $1, 2-3, 4-5$

Rules for Negation

false-elimination

$$\frac{\textbf{false}}{p}\,\textbf{false}\ \mathrm{e}$$

not-elimination

$$\frac{p \quad \neg p}{\textbf{false}}\,\neg\mathrm{e}$$

$$\frac{\boxed{\begin{array}{c} r \\ \vdots \\ \textbf{false} \end{array}}}{\neg r}\,\neg\mathrm{i}$$

From a contradiction, we can prove anything.

Example #7

- Show $p \Rightarrow q, p \Rightarrow \neg q \vdash_{ND} \neg p$

Derived Rule: Proof by Contradiction

Also called RAA (reduction to absurdity).

$$\frac{\begin{bmatrix} \neg r \text{ assumption} \\ \vdots \\ \textbf{false} \end{bmatrix}}{r} \text{ RAA}$$

1	$\neg r \Rightarrow$ **false**	premise
2	$\neg r$	assumption
3	**false**	$\neg$e $1, 2$
4	$\neg\neg r$	$\neg$i $2 - 3$
5	r	$\neg\neg$e 4

Law of the Excluded Middle

$$\frac{}{p \vee \neg p}\ \text{LEM}$$

Summary of Natural Deduction

- Natural deduction for propositional logic is sound and complete.
- A summary of the rules can be found on an additional handout.

Proof Method: Sequent Calculus

- Both natural deduction and the sequent calculus can be found in a paper by Gerhard Gentzen (1909-1945).
  - We will follow the presentation of Fitting and Kelly and use only the connectives: $\wedge, \vee, \Rightarrow$
- **Definition.** A sequent is pair (Γ, Δ) of finite sets of formulas.
- We will write $\Gamma \mapsto \Delta$, and drop the set brackets around the sets of formulas.
  - X will represent a single formula, where Γ as a set of formulas.
- The $\mapsto$ is like implication. A sequent asserts: if all the formulas on the left of the arrow are true, then at least one of the formulas on the right are true.

Meaning of a Sequent

We can extend Boolean valuations to describe the meaning of a sequent.

- $v(\Gamma \mapsto \Delta) = \mathrm{T}$ if $v(X) = \mathrm{F}$ for some $X \in \Gamma$ or $v(Y) = \mathrm{T}$ for some $Y \in \Delta$
- When there is nothing on the LHS or RHS of the arrow, we assume it is the empty set of formulas.
- This means $v(\mapsto) = \mathrm{F}$, and $v(\mapsto X) = v(X)$

Axioms

$$X \mapsto X \quad (Id)$$

$$\text{flase} \mapsto$$

$$\mapsto \text{true}$$

- Note that having axioms we are reminded of Hilbert systems, but this isn't quite the same. $\mapsto$ isn't a symbol in the logic.

Sequent Schemata

The rules of the sequent calculus are written in the form:

$$\frac{S_1}{S_2}$$

Just like in natural deduction,

- if a formula matches the schema S_1,
- then it can be replaced by one matching S_2.

Sequent Calculus Rules

- Structural Rule (Thinning)

  if $\Gamma_1 \subseteq \Gamma_2$ and $\Delta_1 \subseteq \Delta_2$ then : $\dfrac{\Gamma_1 \mapsto \Delta_1}{\Gamma_2 \mapsto \Delta_2}$

  Thinning is like precondition strengthening and postcondition weakening for those familiar with that terminology.
- Adding formulas on the LHS of the sequent is adding them to a conjunction, so this is strengthening the LHS formulas.
- Adding formulas on the RHS of the sequent is adding them to a disjunction, so this is weakening the RHS formulas.

Sequent Calculus Rules

Negation Rules

$$\frac{\Gamma \hookrightarrow \Delta, X}{\Gamma, \neg X \hookrightarrow \Delta} \qquad \frac{\Gamma, X \hookrightarrow \Delta}{\Gamma \hookrightarrow \Delta, \neg X}$$

Conjunction Rules

$$\frac{\Gamma, X, Y \hookrightarrow \Delta}{\Gamma, X \wedge Y \hookrightarrow \Delta} \qquad \frac{\Gamma_1 \hookrightarrow \Delta_1, X \qquad \Gamma_2 \hookrightarrow \Delta_2, Y}{\Gamma_1, \Gamma_2 \hookrightarrow \Delta_1, \Delta_2, X \wedge Y}$$

Disjunction Rules

$$\frac{\Gamma_1, X \hookrightarrow \Delta_1 \qquad \Gamma_2, Y \hookrightarrow \Delta_2}{\Gamma_1, \Gamma_2, X \vee Y \hookrightarrow \Delta_1, \Delta_2} \qquad \frac{\Gamma \hookrightarrow \Delta, X, Y}{\Gamma \hookrightarrow \Delta, X \vee Y}$$

Cut Rule

This a derived rule:

$$\frac{\Gamma_1 \mapsto \Delta_1, X \qquad \Gamma_2, X \mapsto \Delta_2}{\Gamma_1, \Gamma_2 \mapsto \Delta_1, \Delta_2}$$

Proofs in the Sequent Calculus

Definition. A proof is a tree labeled with sequents (generally written with the root at the bottom), such that:

If node N is labeled with $\Gamma \mapsto \Delta$, then if N is a leaf node, $\Gamma \mapsto \Delta$ must be an axiom;

if N has children, their labels must be the premises from which $\Gamma \mapsto \Delta$ follows by one of the rules.

The label on the root node is the sequent that is proved.

Definition. A formula X is a theorem of the sequent calculus if the sequent $\mapsto X$ has a proof, i.e., $\underset{SQ}{\vdash} X$

The sequent calculus for propositional logic is both sound and complete.

Example #1

- Show $\mapsto A \Rightarrow (B \Rightarrow A)$

1 $A \mapsto A$	Id
2 $A, B \mapsto A$	Thinning
3 $A \mapsto B \Rightarrow A$	Implication
4 $\mapsto A \Rightarrow (B \Rightarrow A)$	Implication

Example to Try

- Show $\mapsto \neg(P \wedge Q) \Rightarrow (\neg P \vee \neg Q)$
- Show $\mapsto (\neg A \Rightarrow \neg B) \Rightarrow (B \Rightarrow A)$
- Show $\mapsto (A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$

Sequent Calculus Rules

Implication Rules

$$\frac{\Gamma_1 \hookrightarrow \Delta_1, X \qquad \Gamma_2, Y \hookrightarrow \Delta_2}{\Gamma_1, \Gamma_2, X \Rightarrow Y \hookrightarrow \Delta_1, \Delta_2} \qquad\qquad \frac{\Gamma, X \hookrightarrow \Delta, Y}{\Gamma \hookrightarrow \Delta, X \Rightarrow Y}$$

The left-hand rule is similar to modus ponens. The idea is that if X can be derived, then from $X \Rightarrow Y$, Y can be derived, and therefore whatever can be derived from Y can be derived. If Γ_2, and Δ_1 are empty:

$$\frac{\Gamma_1 \hookrightarrow X \qquad Y \hookrightarrow \Delta_2}{\Gamma_1, X \Rightarrow Y \hookrightarrow \Delta_2}$$

Tableau Proof Methods

see Priest, section 1.5, page 8ff, and Frost, section 4.2.5, page 186ff.

The basis of the Tableau Proof method is to find out whether a formula (to be proven) β leads to a contradiction with a true formula (like a logical axiom or proper axiom) α.

We construct a tree of formulae starting with the initial formulae (the true formula) and the negated conclusion, i.e. the formulae to be proven in negated form.

Successively split up the formulae (according to reverse inference rules) in order to derive simpler formulae.

Similar to resolution, branches of the tree which contain a formula and its negation are closed. They are "cut out".

If every branch of a tableau closes, the tableau is compete and the proof is done.

Tableau Proof

see Priest, section 1.5, page 8ff

Steps to set up the Tableau (as tree):

1. Write assumption and negated conclusion under each other. Formulae underneath each other are conjunctive.
2. Split formulae (if necessary take out implications).
3. Conjunctively joint sub-formulae go into the same branch of the tree, under each other.
4. Disjunctively joined sub-formulae go into separate sub-branches of the tree.
5. Compare atomic propositions in the branches.
6. If a branch contains a proposition and its negation, it will be closed. Write x on its leaves.
7. If all branches are closed, the tableau is complete. Done.

Tableau Proof - Rules (Priest)

$$\begin{tikzcd} & A \supset B \arrow[dl] \arrow[dr] & \\ \neg A & & B \end{tikzcd}$$

$$\begin{tikzcd} \neg(A \supset B) \arrow[d] \\ A \arrow[d] \\ \neg B \end{tikzcd}$$

$$\begin{tikzcd} & A \vee B \arrow[dl] \arrow[dr] & \\ A & & B \end{tikzcd}$$

$$\begin{tikzcd} \neg(A \vee B) \arrow[d] \\ \neg A \arrow[d] \\ \neg B \end{tikzcd}$$

$$\begin{tikzcd} \neg\neg A \arrow[d] \\ A \end{tikzcd}$$

$$\begin{tikzcd} & \neg(A \wedge B) \arrow[dl] \arrow[dr] & \\ \neg A & & \neg B \end{tikzcd}$$

$$\begin{tikzcd} A \wedge B \arrow[d] \\ A \arrow[d] \\ B \end{tikzcd}$$

$$\begin{tikzcd} & A \equiv B \arrow[dl] \arrow[dr] & \\ A \arrow[d] & & \neg A \arrow[d] \\ B & & \neg B \end{tikzcd}$$

$$\begin{tikzcd} & \neg(A \equiv B) \arrow[dl] \arrow[dr] & \\ A \arrow[d] & & \neg A \arrow[d] \\ \neg B & & B \end{tikzcd}$$

Tableau Proof (example 1)

$$\begin{tikzcd}
& & A \supset B \arrow[d] & & \\
& & B \supset C \arrow[d] & & \\
& & \neg(A \supset C) \arrow[d] & & \\
& & A \arrow[d] & & \\
& & \neg C \arrow[dl] \arrow[dr] & & \\
& \neg A \arrow[dl] \arrow[d] & & B \arrow[d] \arrow[dr] & \\
\neg B & C & & \neg B & C \\
\times & \times & & \times & \times
\end{tikzcd}$$

Tableau Proof (example 2)

$$\begin{tikzcd}
& & \neg(((A \supset B) \wedge (A \supset C)) \supset (A \supset (B \wedge C))) & & & \\
& & (A \supset B) \wedge (A \supset C) & & & \\
& & \neg(A \supset (B \wedge C)) & & & \\
& & (A \supset B) & & & \\
& & (A \supset C) & & & \\
& & A & & & \\
& & \neg(B \wedge C) \arrow[dl] \arrow[dr] & & & \\
& \neg B \arrow[dl] \arrow[d] & & \neg C \arrow[d] \arrow[dr] & & \\
\neg A & B & & \neg A & B \arrow[d] \arrow[dr] & \\
\times & \times & & \times & \neg A & C \\
& & & & \times & \times
\end{tikzcd}$$

Tableau Proof with Inverse Sequent Rules (Frost)

see Frost, section 4.2.5, page 186ff.

Steps to set up the Tableau (as table):

1. Write true formula α and to be proven conclusion β on two sides of a column. (α left and β right)
2. Apply inverse sequent logic rules ($\cong$ inverse inference rules).
    1) E.g. substitute $P \land Q$ by P and Q in same column.
    2) E.g. substitute $R \Rightarrow S$ on the right side by adding R to the left side and S to the right side.
3. If the same atomic formula appears on both sides, the tableau closes. Done.

Resolution Principle
- a powerful inference rule

- Literal: either an atom (positive literal) or the negation of an atom (negative literal).
- Clause: $(P_1 \lor P_2 \lor \ldots \lor P_n)$
- A wff can be represented as a set of clauses
    - first translate to conjunctive normal form

Example:

$(P \lor Q \lor \neg R) \land (P \lor Q \lor \neg R) \rightarrow \{(P \lor Q \lor \neg R), (P \lor Q \lor \neg R)\}$

Note: *Empty clause {} is equivalent to false.*

Resolution principle
- Converting wffs to set of clauses

Example: $\neg(P \Rightarrow Q) \vee (R \Rightarrow P)$

1.	$\neg(\neg P \vee Q) \vee (\neg R \vee P)$	Definition of $\vee$
2.	$(P \wedge \neg Q) \vee (\neg R \vee P)$	DeMorgan
3.	$(P \vee \neg Q \vee P) \wedge (\neg Q \vee \neg R \vee P)$	Distributivity
4.	$(P \vee \neg R) \wedge (\neg Q \vee \neg R \vee P)$	Associativity

Usually expressed as $\{(P \vee \neg R), (\neg Q \vee \neg R \vee P)\}$

Resolution Principle

resolvent

Given clause sets $\Sigma_1 \cup \Sigma_2$,

$$\lambda \vee \Sigma_1 , \neg\lambda \vee \Sigma_2 \vdash \Sigma_1 \vee \Sigma_2$$

Examples:

- $R \vee P, \neg P \vee Q \vdash R \vee Q$: chaining
- $R, \neg R \vee P \vdash P$: modus ponens
- Chaining and modus ponens are special cases of resolution principle.
- $P \vee Q \vee R \vee S, \neg P \vee Q \vee W \vdash Q \vee R \vee S \vee W$

Resolution principle
- on Clauses

$P \vee Q \vee \neg R$, $P \vee W \vee \neg Q \vee R$

- Resolving them on Q: $P \vee \neg R \vee R \vee W$
- Resolving them on R: $P \vee \neg Q \vee Q \vee W$
- Since $\neg R \vee R$ and $\neg Q \vee Q$ are *True*, the value of each of these resolvents is *True*.
- We must resolve either on Q or on R.
- $P \vee W$ is not a resolvent of the two clauses.

Resolution principle
- Soundness

$$\lambda \vee \Sigma_1 \text{ , } \neg\lambda \vee \Sigma_2 \vdash \Sigma_1 \vee \Sigma_2$$

Proof: reasoning by cases

- Case 1: λ is *true*
  - Σ_2 must *true* in order for $\{\neg\lambda\} \cup \Sigma_2$ to be *true*.
- Case 2: λ is *false*,
  - Σ_1 must *true* in order for $\{\neg\lambda\} \cup \Sigma_1$ to be *true*.
- Either Σ_1 or Σ_2 must be true.
- $\Sigma_1 \vee \Sigma_2$ must be true.

Resolution principle - incompleteness

Resolution principle is not complete.

- For example, $P \wedge R \not\models P \vee R$
- We cannot infer $P \vee R$ using resolution on the set of clauses $\{P, R\}$ (because there is nothing that can be resolved)
- We cannot use resolution directly to check all logical entailments.

Resolution principle - proof by refutation $\Delta \models \omega$

Instead, we refute $\Delta \wedge \neg\omega$

1. Convert a wff Δ to set of clauses.
2. Convert $\neg\omega$ to a clause.
3. Let Γ bet the set of clauses from steps 1 and 2.
4. Iteratively apply the principle to the clauses in Γ and add the results to Γ until either
   - no more resolvents can be added or
   - an empty clause is produced.

Resolution principle
- Completeness of Refutation

- { } will be produced if $\Delta \models \omega$.
- Resolution principle is *refutation complete* for propositional calculus.
- Decidability

  If
  - Δ is a finite set of clauses and
  - $\Delta \not\models \omega$,

  then resolution principle terminates without {}.

111

Work out

- Please submit one sheet in which you argue why resolution principle always terminate with propositional calculus.

112

Resolution principle
- a refutation tree

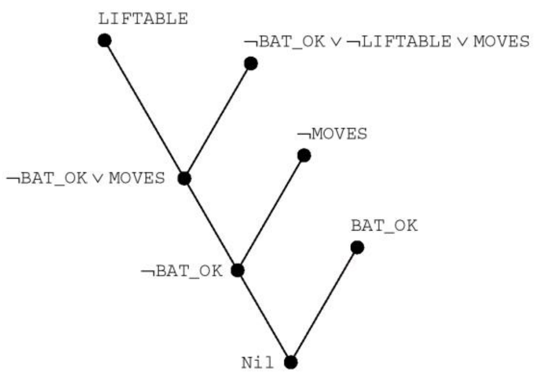


Figure 14.1 A Resolution Refutation Tree

Given:
1. BAT_OK
2. ¬MOVES
3. BAT_OK ∧ LIFTABLE ⊃ MOVES

Clause form of 3:

4. ¬BAT_OK ∨ ¬LIFTABLE ∨ MOVES

Negation of goal:

5. LIFTABLE

Perform resolution:

6. ¬BAT_OK ∨ MOVES (from resolving 5 with 4)
7. ¬BAT_OK (from 6, 2)
8. Nil (from 7, 1)

Resolution principle
- refutations search strategies

- Ordering strategies
  - Breadth-first strategy
  - Depth-first strategy
    - with a depth bound, use backtracking.
  - Unit-preference strategy
    - prefer resolutions in which at least one clause is a unit clause.
- Refinement strategies
  - Set of support
  - Linear input
  - Ancestry filtering

Resolution principle
- refinement strategies

- Set of support strategy
  Refutation complete
  - Allows only those resolutions in which one of the clauses being resolved is in the set of support,
  - i.e., those clauses that are either clauses coming from the negation of the theorem to be proved or descendants of those clauses.
- Linear input strategy
  Not refutation complete
  - at least one of the clauses being resolved is a member of the original set of clauses.
- Ancestry filtering strategy
  Refutation complete
  - at least one member of the clauses being resolved either is a member of the original set of clauses or is an ancestor of the other clause being resolved.
  - Refutation complete

Workout

- Please give an example set of clauses that shows the linear input strategy makes resolution principle incomplete with resolution principle.

Resolution principle
- Horn clauses

- A Horn clause: a clause that has at most one positive literal.
- Ex: P, $\neg P \vee Q$, $\neg P \vee \neg Q \vee R$, $\neg P \vee \neg R$
- Three types of Horn clauses.
  - A single atom: called a "fact"
  - An implication: called a "rule"
  - A set of negative literals: called "goal"
- There are linear-time deduction algorithms for propositional Horn clauses.

117

BDD (Binary Decision Diagram)
- Overview

- Decision trees and reduction rules
- Building already reduced diagrams
- Dynamic variable reordering
- BDD operators
- Implementations issues
- (Dis)advantages of BDDs

118

Binary Decision Diagram (BDD)

unique BDD for a function

- BDD: A *minimal canonical* form ntation for Boolean formulas.

minimum size representatio n (for a given var ordering)

on:

  - uch space redundancy in traditional presentations
  - BDD is more compact than truth tables, conjunctive normal form, disjunctive normal form, binary decision trees, etc.
  - BDD has a canonical form
  - BDD operations are efficient

BDD (Binary Decision Tree)

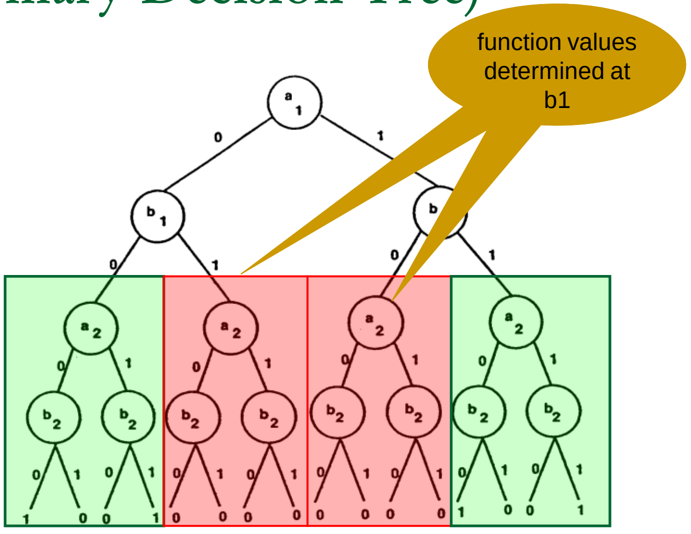


Figure 5.1
Binary decision tree for two-bit comparator.

BDD (redundancy in BDT)

- Binary Decision Trees (BDT):
  - Same size as truth tables
  - Lots of redundancy: Out of 8 subtrees rooted at b_2 only 3 are distinct!!!
  - Merge isomorphic subtrees ➔ BDD
- BDD is a rooted, DAG with 2 types of vertices: terminal and nonterminal
- Each nonterminal *v* is labeled with var(*v*) and has two successors: low(*v*) and high(*v*)
- Each terminal vertex is labeled 0 or 1

Truth Table, DNF, and CNF

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

- DNF (sum-of-products)

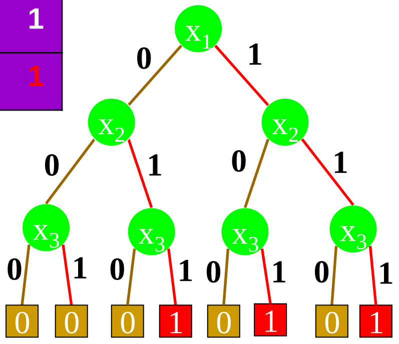
$F = x_1'x_2x_3 + x_1x_2'x_3 + x_1x_2x_3$

- CNF (product-of-sums)

$F = (x_1+x_2+x_3)\ \&\ (x_1+x_2+x_3')\ \&$
$(x_1+x_2'+x_3)\ \&\ (x_1'+x_2+x_3)\ \&$
$(x_1'+x_2'+x_3)$

Truth Table and Decision Tree

x_1	0	0	0	0	1	1	1	1
x_2	0	0	1	1	0	0	1	1
x_3	0	1	0	1	0	1	0	1
F	0	0	0	1	0	1	0	1



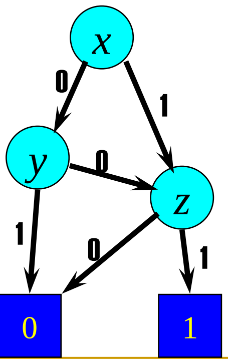
BDD (Binary Decision Diagram)

minimal canonical form of Boolean functions

- minimal → efficiency (space & computation)
  - for a specific variable ordering
- Canonical form:
  unique BDD for a function

$$(x \vee \neg y) \wedge z$$

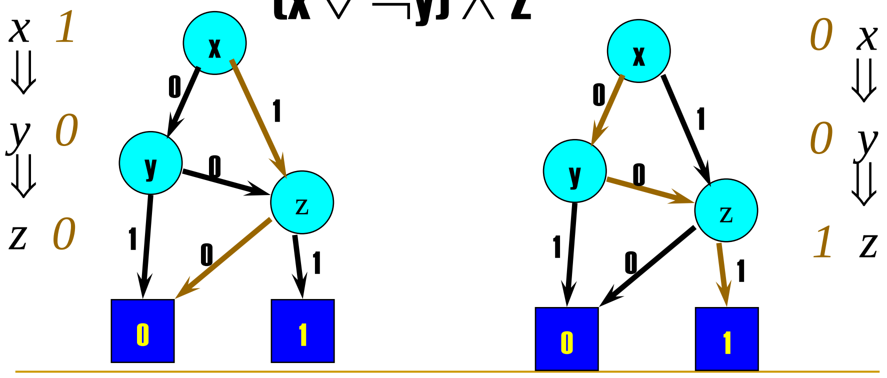
0 0 1



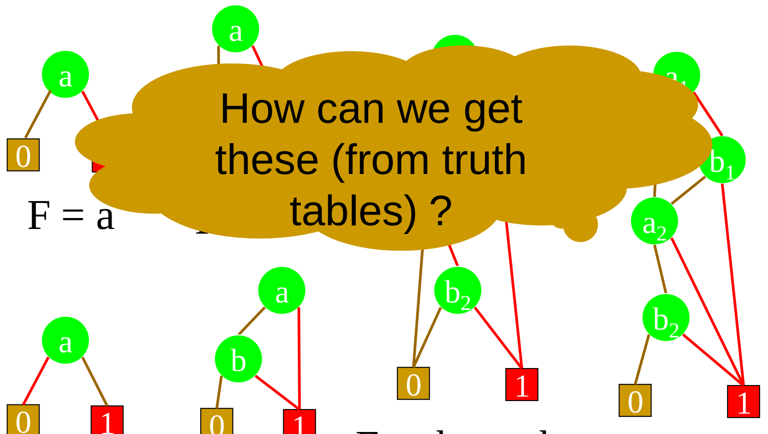
BDD function evluation

Using path traversal to evaluate function

$(x \vee \neg y) \wedge z$



BDD Examples



F = a

F = a′

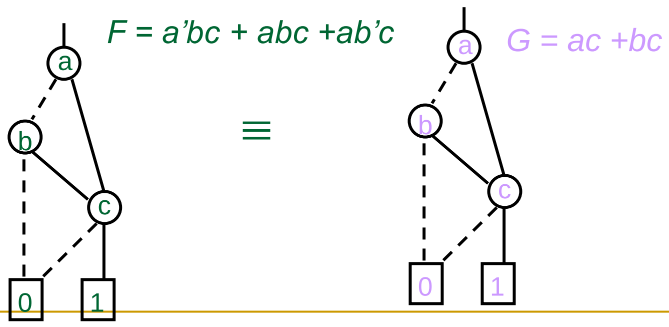
F = a+b

$F=a_1b_1+ a_2b_2$

$F=a_1b_1+a_2+b_2$

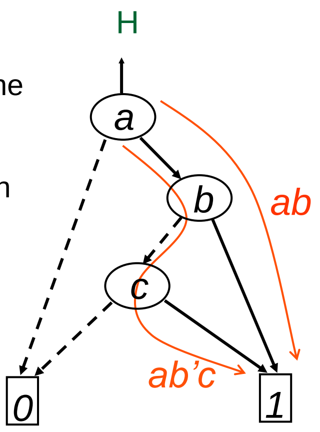
Application to Verification

- Equivalence of *combinational* circuits
- *Canonicity* property of BDDs:
  - if F and G are equivalent, their BDDs are identical (for the same ordering of variables)



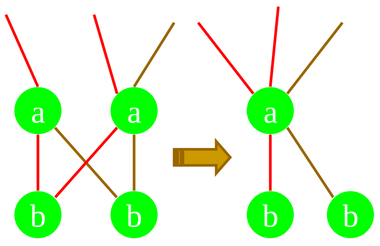
Application to Verification, cont'd

- Functional test generation
  - SAT, Boolean *satisfiability* analysis
  - to test for H = 1 (0), find a path in the BDD to terminal **1** (0)
  - the path, expressed in function variables, gives a satisfying solution (test vector)

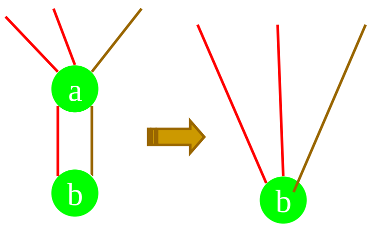


Reduction of Decision Tree

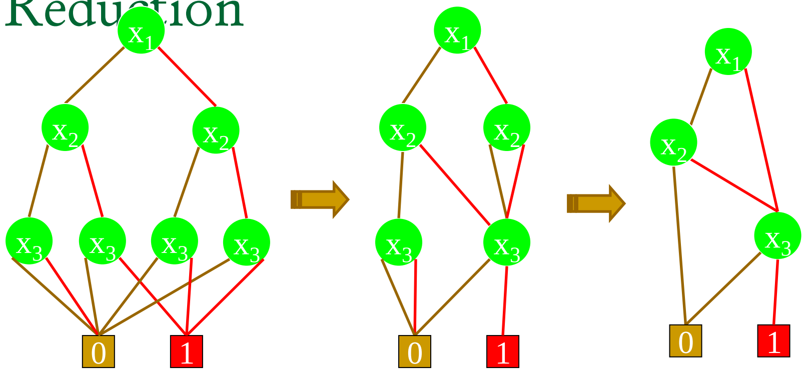
Rule 1: Merging Rule:
Nodes must be unique



Rule 2: Elimination Rule:
Redundant tests should not be present



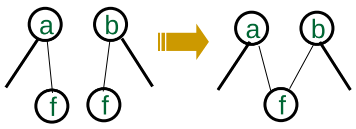
Example of Decision Tree Reduction

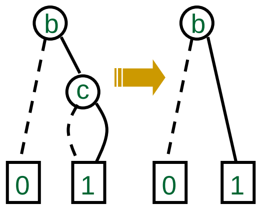


Decision tree reduction BDD

BDD Construction

- Typically done using *APPLY* operator
- Reduction rules
  - remove duplicate terminals
  - merge duplicate nodes (*isomorphic* subgraphs)
  - remove *redundant* nodes
- Redundant nodes:
  - nodes with identical children





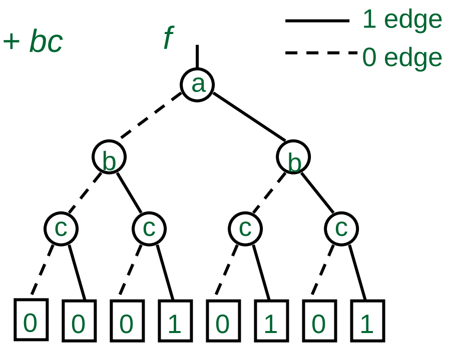
BDD Construction – your first BDD

- Construction of a Reduced Ordered BDD

f = ac + bc

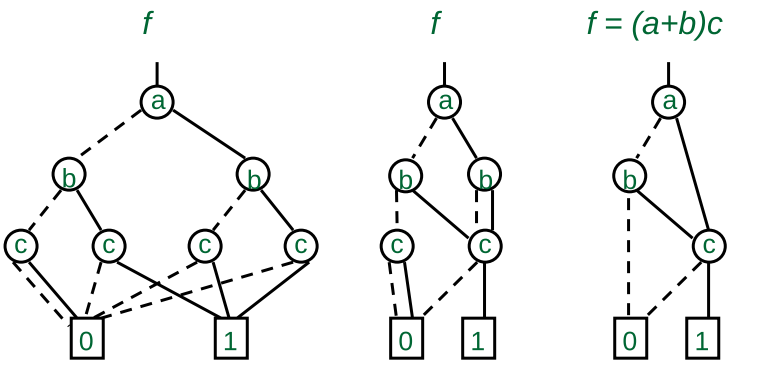
a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Truth table



Decision tree

BDD Construction – cont'd

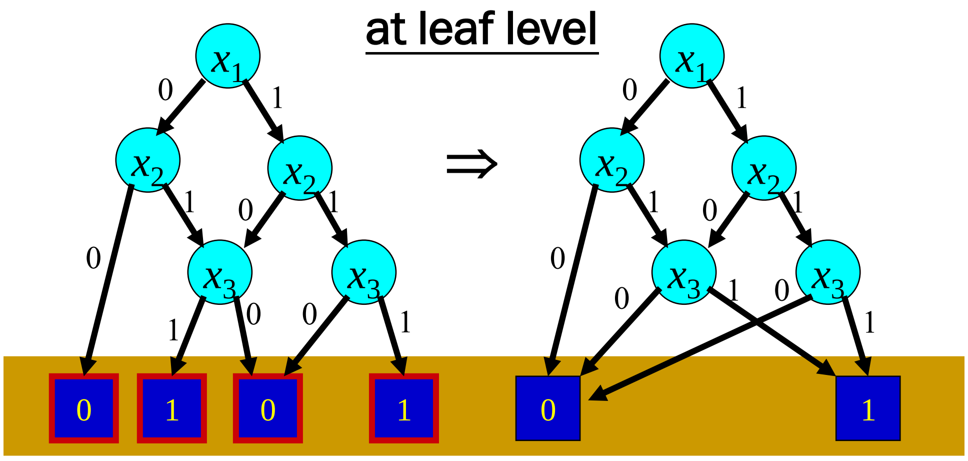


1. Remove duplicate terminals

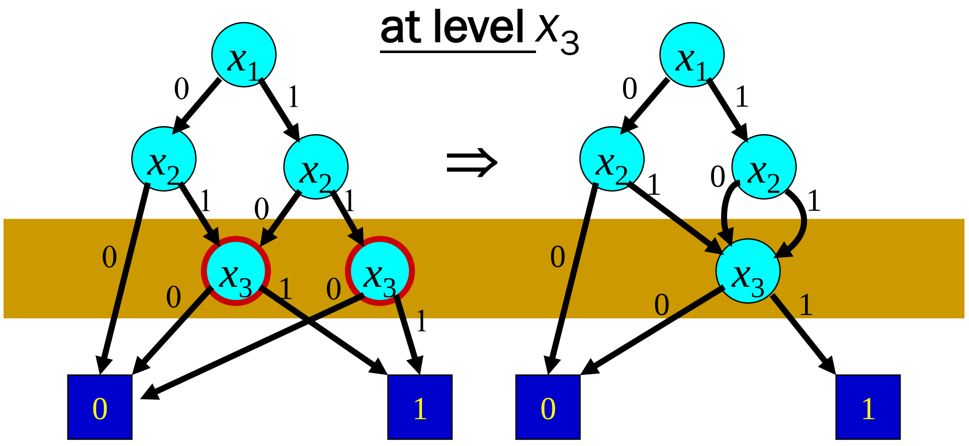
2. Merge duplicate nodes

3. Remove redundant nodes

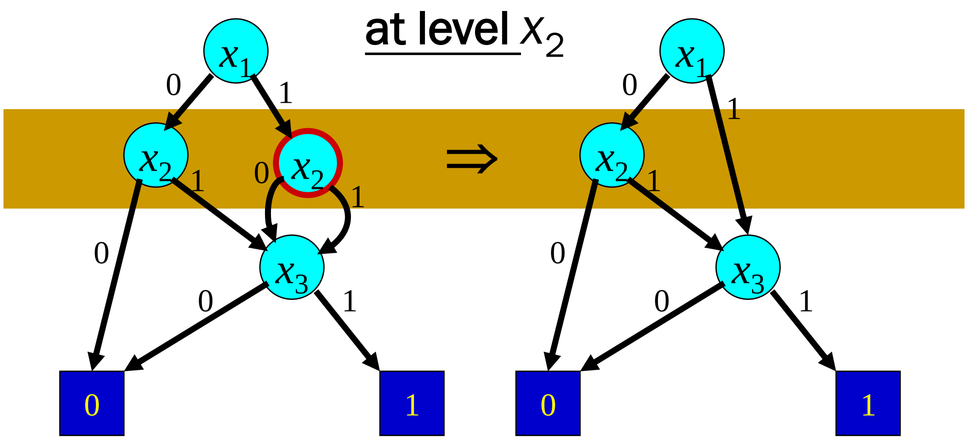
BDD Reduction (I) - from bottom up



BDD Reduction (II) - from bottom up

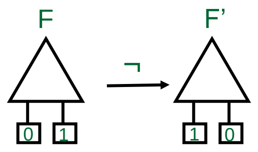


BDD Reduction (III) - from bottom up

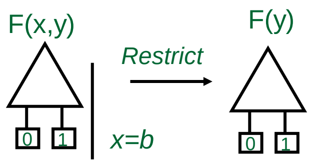


Logic Manipulation using BDDs

- Useful operators
  - *Complement* $\neg$ *F = F'*
    (switch the terminal nodes)

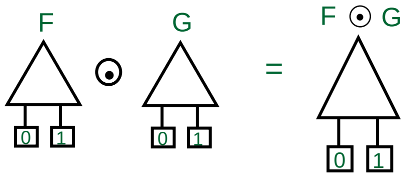


  - *Restrict: F*$|_{x=b}$ = *F(x=b)*
    where *b* = const



Useful BDD Operators - cont'd

- *Apply: F* ⊙ *G*
  where ⊙ stands for any Boolean operator (AND, OR, XOR, →)



- Any logic operation can be expressed using only *Restrict* and *Apply*
- Efficient algorithms, work directly on BDDs

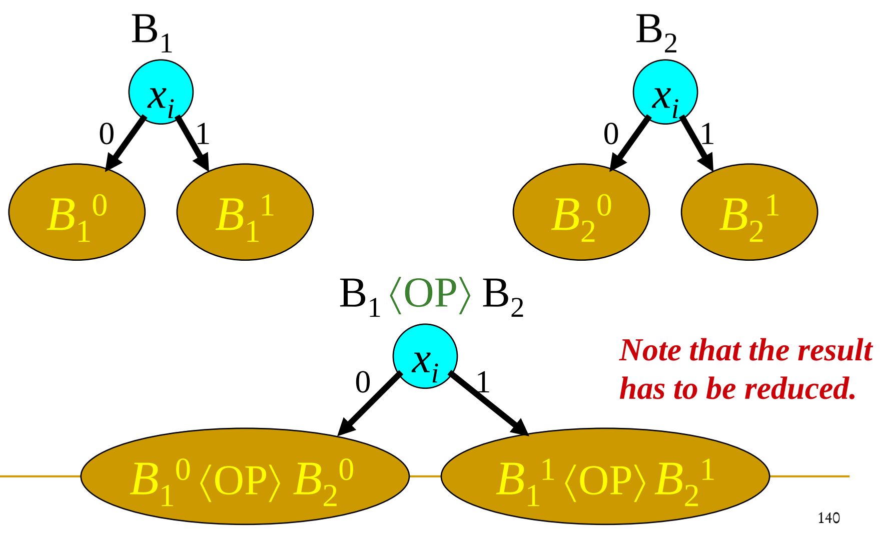
Apply Operation

- Basic operator for efficient BDD manipulation (structural)
- Based on recursive *Shannon expansion*

$$\boxed{F \text{ OP } G = x\,(F_x \text{ OP } G_x) + x'(F_{x'} \text{ OP } G_{x'})}$$

where *OP = OR, AND, XOR, etc*

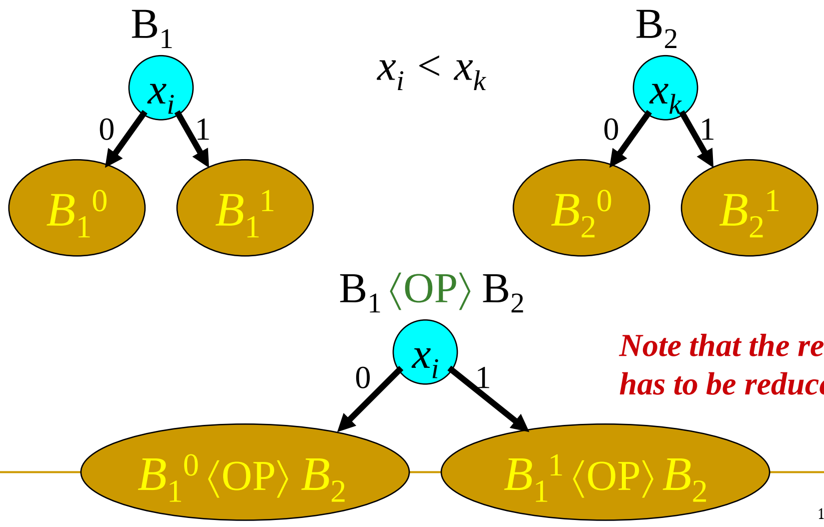
$B_1 \langle OP \rangle B_2$



B_1
x_i
0
1
B_1^0
B_1^1
B_2
x_i
0
1
B_2^0
B_2^1
$B_1 \langle OP \rangle B_2$
x_i
0
1
$B_1^0 \langle OP \rangle B_2^0$
$B_1^1 \langle OP \rangle B_2^1$

Note that the result has to be reduced.

$B_1 \langle OP \rangle B_2$ - case II

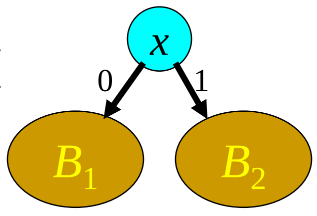

B_1
x_i
0
1
B_1^0
B_1^1
$x_i < x_k$
B_2
x_k
0
1
B_2^0
B_2^1
$B_1 \langle OP \rangle B_2$
x_i
0
1
$B_1^0 \langle OP \rangle B_2$
$B_1^1 \langle OP \rangle B_2$

Note that the result has to be reduced.

$B_1 \langle OP \rangle B_2$

bdd_root(x,B_1,B_2) {

return(reduce(



));

}

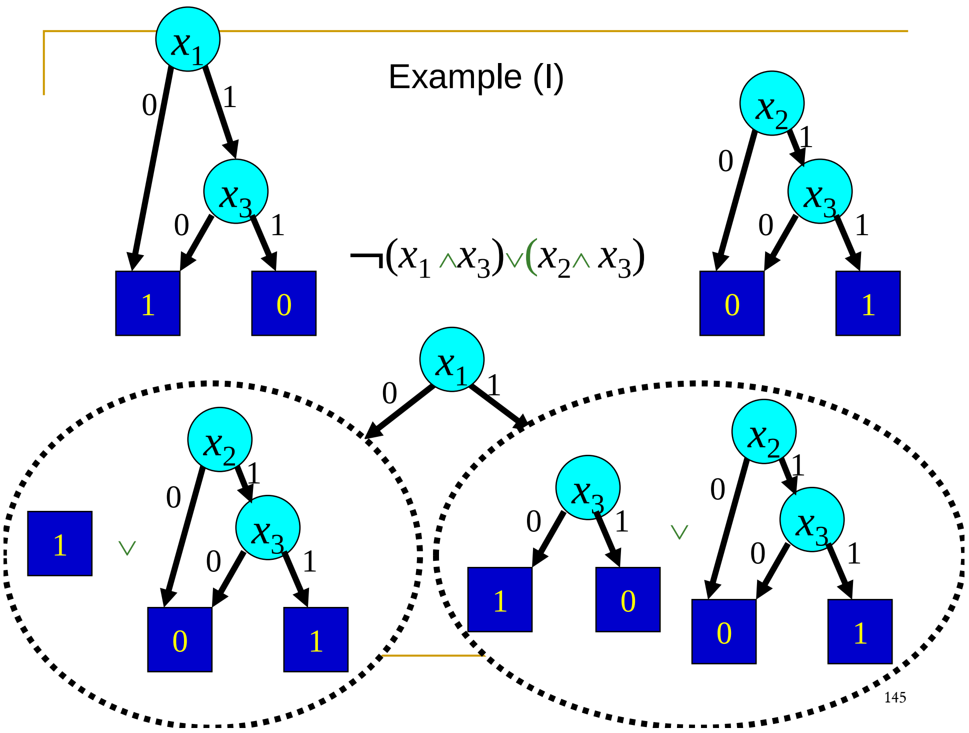
$B_1 \langle OP \rangle B_2$ - procedure template

```
op(B_1,B_2) {
  if the pair has already been processed, return the saved result.
…/* case for terminal nodes */
  else if (root(B_1)== root(B_2)) {
     return(reduce(bdd_root(
        root(B_1), op(B^0_1,B^0_2), op(B^1_1,B^1_2)
     )));
  }
  else …/* case for root(B_1)!= root(B_2)*/
  save the result.
}
```

$B_1 \langle OP \rangle B_2$ - procedure template

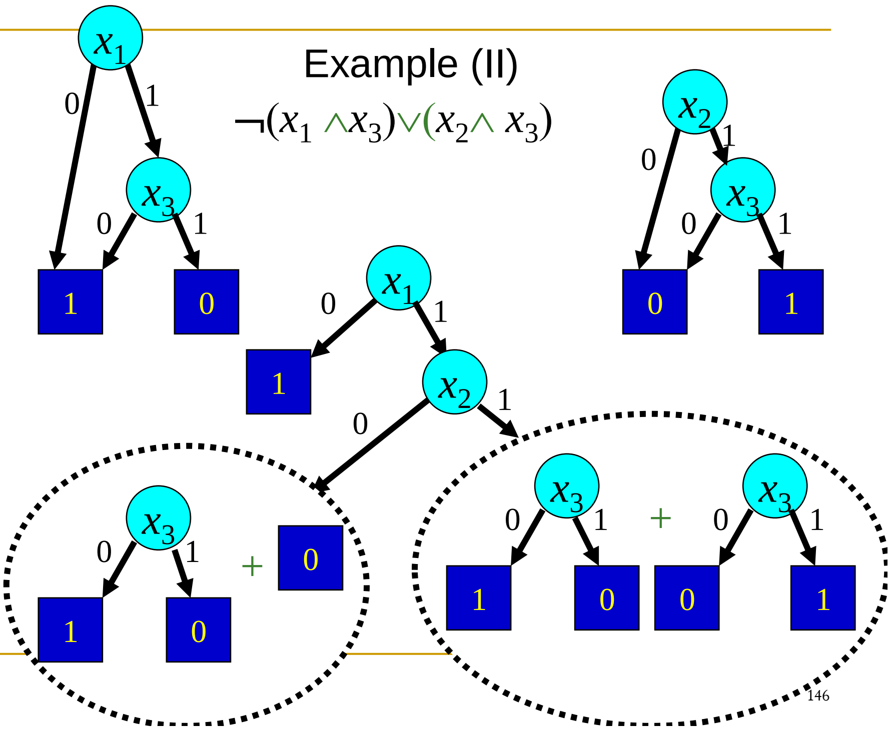
```
op(B_1,B_2) {
  if the pair has already been processed, return the saved result.
  …/* case for terminal nodes */
  else if (root(B_1)< root(B_2)) {
     return(reduce(bdd_root(
        root(B_1), op(B^0_1,B_2), op(B^1_1,B_2)
     )));
  }
  else …/* case for root(B_1)>= root(B_2)*/
  save the result.
}
```

Example (I)



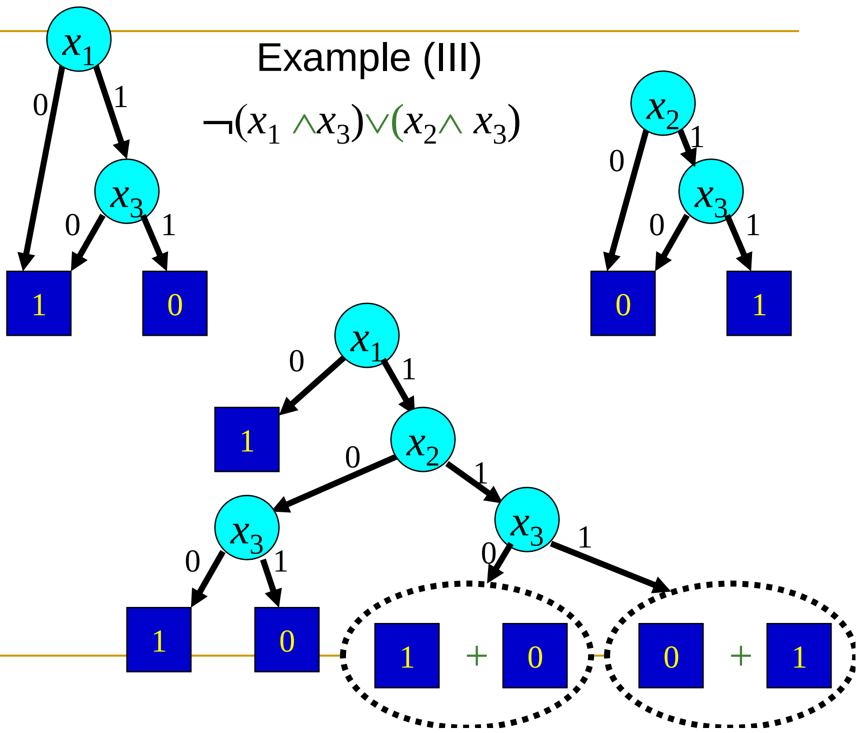
Example (II)

$\neg(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$



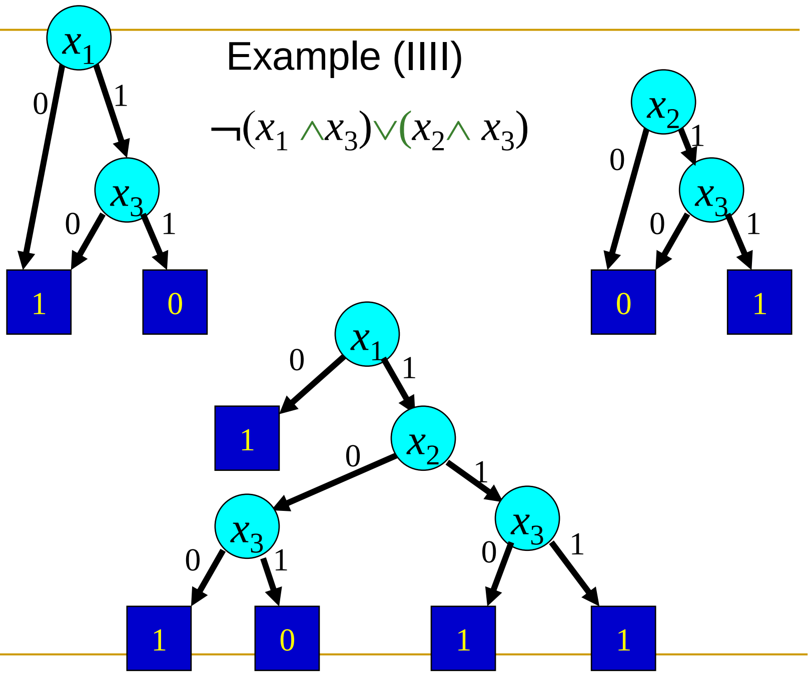
Example (III)

$\neg(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$



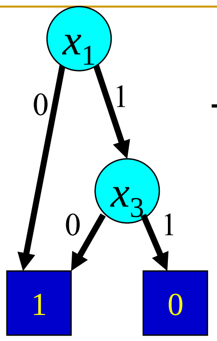
Example (IIII)

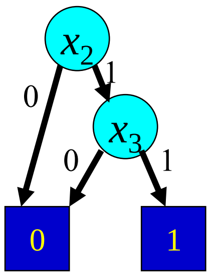
$\neg(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$

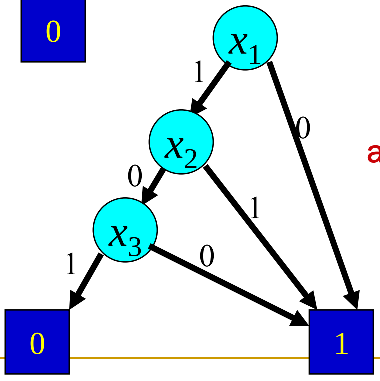


Example (V)

$\neg(x_1 \wedge x_3) \vee (x_2 \wedge x_3)$





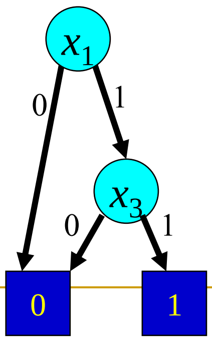


after reduction

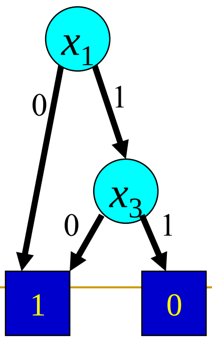
¬B

Exchange $\boxed{0}$ with $\boxed{1}$

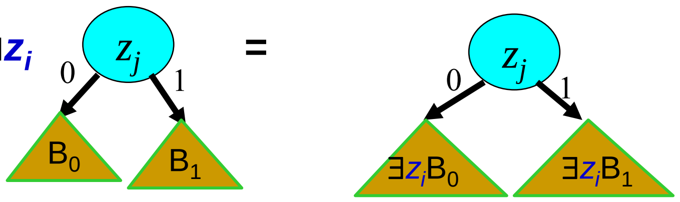
$x_1 x_3$

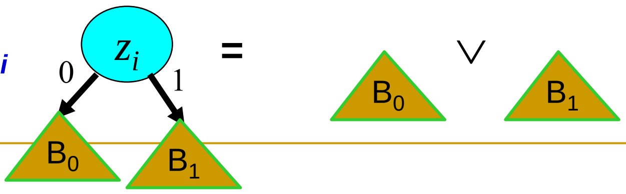


$\neg(x_1 x_3)$



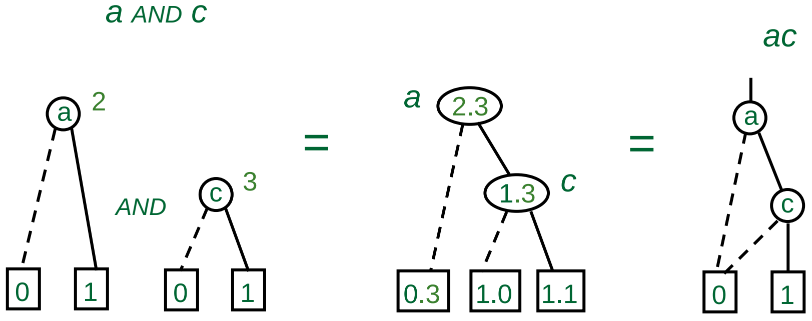
$\exists z_i\, B(z_1, \ldots\ldots, z_n)$

- $\exists z_i\ \boxed{0} = \boxed{0}$; $\exists z_i\ \boxed{1} = \boxed{1}$;
- $\exists z_i$ 

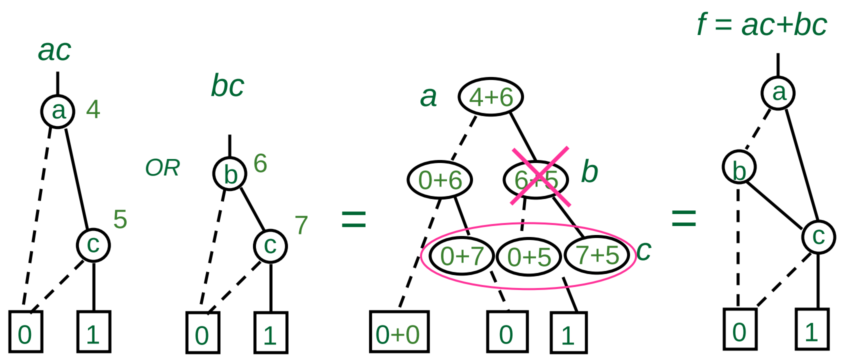
- $\exists z_i$ 

Apply Operation - *AND*

a AND *c*



Apply Operation - *OR*



Building Already Reduced Tree

assuming variable ordering

$x_1 < x_2 < x_3 < \ldots < x_n$

```
function Build ( F, i )
if ( i  > n )
        if (F == 0)    return Node0
        else           return Node1;
else
        v0 = Build ( F(xi=0), i+1 );
        v1 = Build ( F(xi=1), i+1 );
        return  CreateNode ( i, v0, v1 );
```

Creating a Unique Node

assuming there is a node table and
functions CheckExists(), Insert(), and Hash()

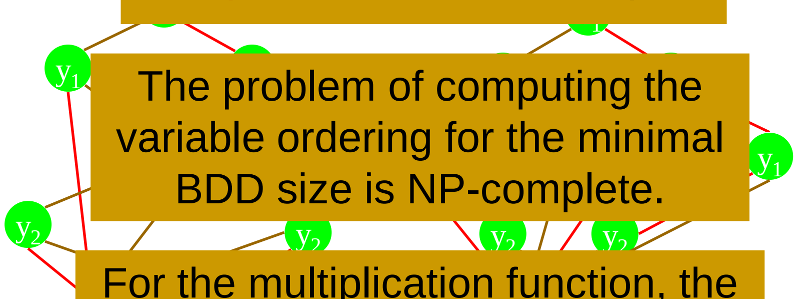
```
function CreateNode( Var, LowF, HighF )
if ( LowF == HighF )
        return LowF;
else if ( CheckExists( Var, LowF, HighF ) )
        return the existing node;
else   Insert( Var, LowF, HighF );
        return the new node;
```

Variable Ordering

$F(x_1,$ $)$

$x_1 < y$ y_2

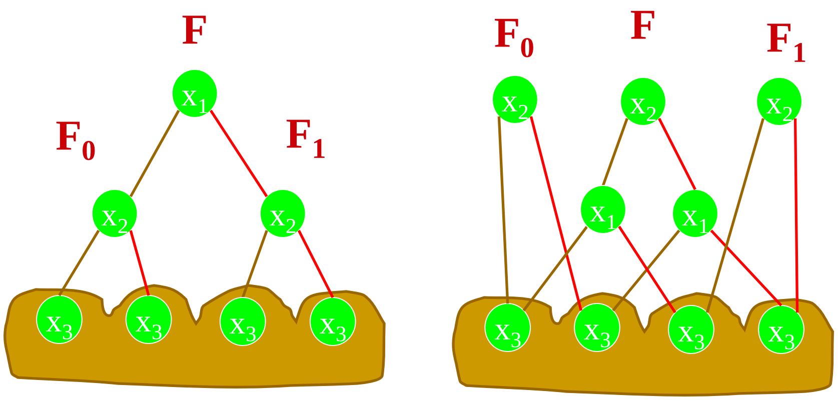
Subject to the variable orderings, BDD may have *exponential* size blow-up.



The problem of computing the variable ordering for the minimal BDD size is NP-complete.

For the multiplication function, the circuit is always exponential.

Dynamic Variable Reordering



Variable reordering is a local operation

BDD manipulation complexities

Reduce(B)	O(|B|log|B|)
$B_1 \langle OP \rangle B_2$	$O(\|B_1\|\|B_2\|)$
¬B	O(1)
CreateNode	O(1)
Build	$O(2^n)$
APPLY	O(|F|*|G|)
RESTRICT	O(|F|)
COMPOSE	$O(\|F\|^2 *\|G\|^2)$

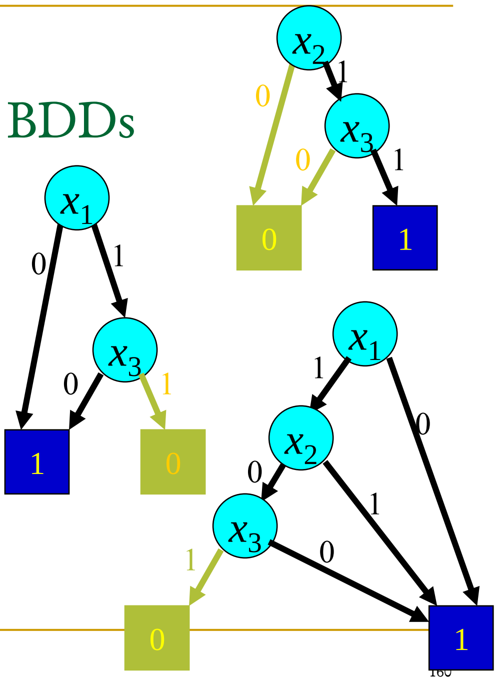
Workout for BDD

1. Please draw the BDD of $((\neg x_1) \vee x_3) \wedge (x_2 \vee x_3)$.
2. Please draw the BDD of

   $(\neg(x_1 \wedge x_3) \vee (x_2 \wedge x_3)) \vee (((\neg x_1) \vee x_3) \wedge (x_2 \vee x_3))$

BDD variations - Single-terminal BDDs

- By omitting 0, we still have a representation of the function.
- When there is no branches for a truth assignment, then the value is FALSE (0).
- Save memory!

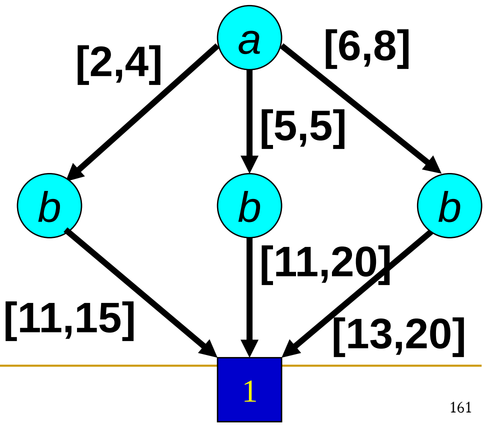


BDD variations
- Multi-terminal BDDs (MTBDDs)

discrete a, b:0..100.

$(2 \le a \le 5 \wedge b \in [11,15]) \vee (a \in [4,8] \wedge 13 \le b \le 20)$

- Using decimals instead of binaries.
- More intuitistic.
- Complexer to implement.



Characteristic Functions: Example

- Problem: Given the set {p_1, p_2, p_3, p_4, p_5, p_6 }, create the characteristic function of the subset {p_1, p_3, p_4} and represent it using BDDs
- Step 1: Introduce an encoding of the set

p_1 – $\bar{x}_2\bar{x}_1\bar{x}_0$ p_2 – $\bar{x}_2\bar{x}_1x_0$ p_3 – $\bar{x}_2x_1\bar{x}_0$

p_4 – $\bar{x}_2x_1x_0$ p_5 – $x_2\bar{x}_1\bar{x}_0$ p_6 – $x_2\bar{x}_1x_0$

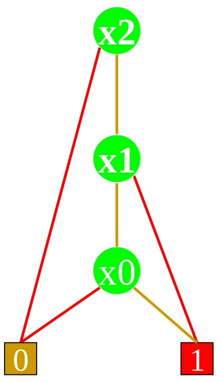
- Step 2: Define a function over the encoding variables (x_1, x_2, x_3) such that it will be equal to 1 for minterms encoding the subset {p_1, p_3, p_4}.

Characteristic Functions: Example (continued)

$\varphi_{\{p1,\ p3,\ p4\}}$ (x_0, x_1, x_2) =

= $\bar{x}_2\bar{x}_1\bar{x}_0$ + $\bar{x}_2 x_1\bar{x}_0$ + $\bar{x}_2 x_1 x_0$

- Step 3: Represent this function as a BDD



Set Manipulation

Operations on combinatorial sets can be reduced to boolean operations on characteristic functions

Empty set: $\chi_{\varnothing} = 0$

Union of sets: $\chi_{S \cup T} = \chi_S + \chi_T$

Intersection of sets: $\chi_{S \cap T} = \chi_S \,\&\, \chi_T$

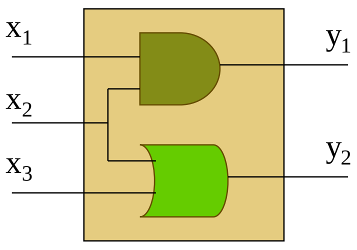
Difference of sets: $\chi_{S - T} = \chi_S \,\&\, \chi_T'$

Subset relation ($S \subset T$): $\chi_{S - T} = \chi_S \,\&\, \chi_T' = 0$

Relations for Logic Blocks

- Suppose variables ($x_1,x_2,x_3,...$) and ($y_1,y_2,y_3,$) are inputs and outputs of a logic block.
- Then, we can define a relation over variables ($x_1,x_2,x_3,...$) and ($y_1,y_2,y_3,...$). Suppose variables are related in the following way. Assignments ($y_1,y_2,y_3,...$) correspond to values, which outputs take when values ($x_1,x_2,x_3,...$) are applied at the inputs.

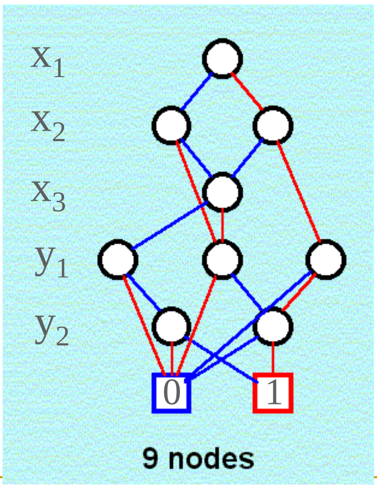
Relations for Logic Blocks: Example



x_1	x_2	x_3	y_1	y_2
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

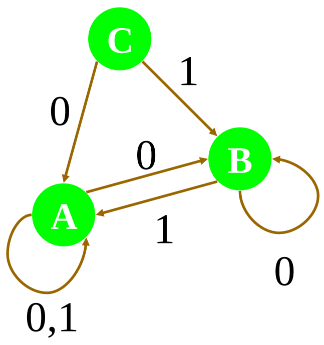
Example (continued)

x_1	x_2	x_3	y_1	y_2	F
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	0	1	1
1	0	0	0	0	1
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	1	1
other					0



Relations for FSMs: Example

Ins	CS	CSC	NS	NSC
0	A	00	B	10
0,1	A	00	A	00
0	B	10	B	10
1	B	10	A	00
0	C	01	B	10
1	C	01	A	00



Example (continued)

Relation =
- $i'a_1`a_2`b_1b_2`+$
- $a_1`a_2'b_1'b_2' +$
- $i'a_1a_2'b_1b_2' +$
- $ia_1a_2'b_1'b_2' +$
- $i'a_1a_2'b_1b_2' +$
- $ia_1a_2'b_1'b_2'$

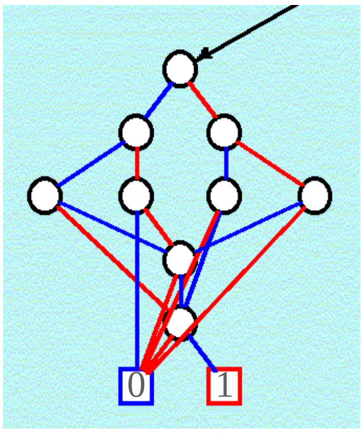
i

a_1

b_1

a_2

b_2



Efficient Operations on BDDs

- Apply – NOT, AND, OR, EXOR, etc.
- Quantification (existential, universal)
- Replace
- Compose
- Specialized operators
  - Generalized cofactor (constrain, restrict)
  - Compatible projection, etc.

Components of BDDs and Their Use

- Nodes (represent functions; complexity)
- Terminal nodes (constant functions)
- Edges (relationship between functions)
- Paths (true and false var. assignments)
- Cuts (variable partitions and subsets)

171

Properties of BDDs

- Canonicity
- Compactness (with some exceptions)
- Represent a variety of discrete objects
  - Boolean functions
  - Compositional sets
  - Encodings and labelings
- Facilitate symbolic methods
  - Two-level minimization
  - State traversal of FSMs
  - ...

172

(Dis)Advantages of BDDs

- Universal (for discrete data only)
- Save memory (not always)
- Speed-up computation (not always)
- Attractive coding style (it depends...)
- Implicit computation (what about good old classical methods?)

Overview

- ITE operator
- APPLY operator
- RESTRICT operator
- Derived operators
- The effect of variable ordering
- Deriving the Upper Bound on BDD Size for Boolean Functions
- Dynamic variable reordering

IF-THEN-ELSE (ITE) Operator

- Boolean operations over 2 arguments can be expressed as ITE of F, G, and constants

  ITE(F, G, H) = F & G + F′ & H
- Example: AND(F, G) = ITE(F, G, 0)
- Computation of boolean operations is based on the Shannon expansion

  $\text{ITE}(F,G,H) = \text{ITE}(x, \text{ITE}(F_{x'},G_{x'},H_{x'}),$
  $\text{ITE}(F_x,G_x,G_x))$

APPLY operator

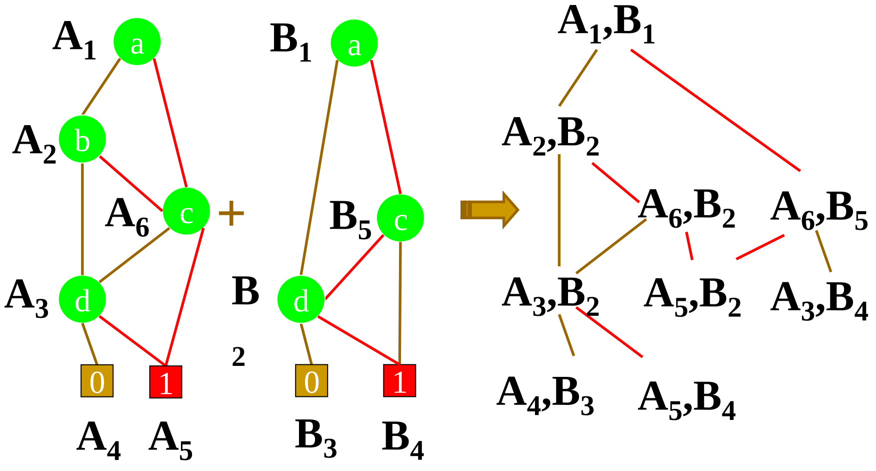
- APPLY(F, G) operator is a shorthand for any two-variable boolean operator
- APPLY is reducible to ITE
- It follows that APPLY can be computed recursively just like ITE

  APPLY(F,G) = x′ & APPLY(Fx′,Gx′) +
  x & APPLY(Fx ,Gx)

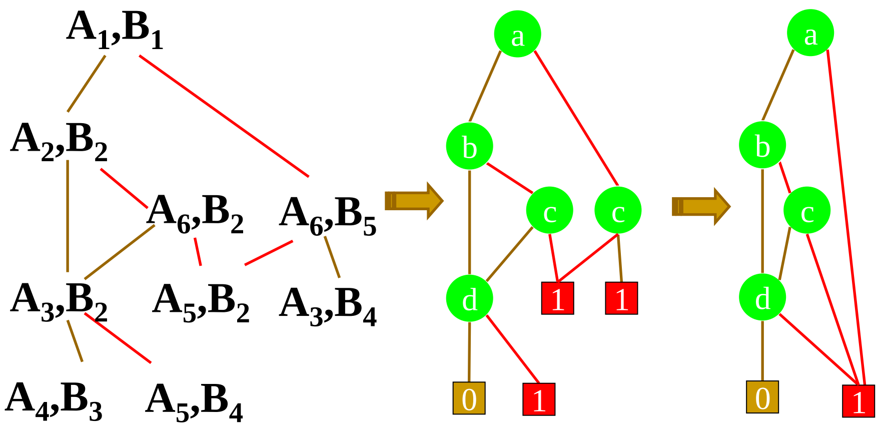
Pseudocode for APPLY operator

```
function Apply( F, G )
if ( AlreadyComputed( F, G ) ) return the result;
else if ( F=={0,1} && G=={0,1} ) return oper( F, G );
else if ( Var( F ) == Var( G ) )
    u = CreateNode( Var(F), Apply(Fx′,Gx′), Apply(Fx,Gx));
else if ( Var( F ) < Var( G ) )
    u = CreateNode( Var(F) , Apply(Fx′,G ), Apply(Fx,G ));
else /* if ( Var( F ) > Var( G ) ) */
    u = CreateNode( Var(F) , Apply(F,Gx′ ), Apply(F,Gx ));
    InsertComputed( F,G,u );
return u;
```

F=ac+bc+d G=ac′+d F+G = ?



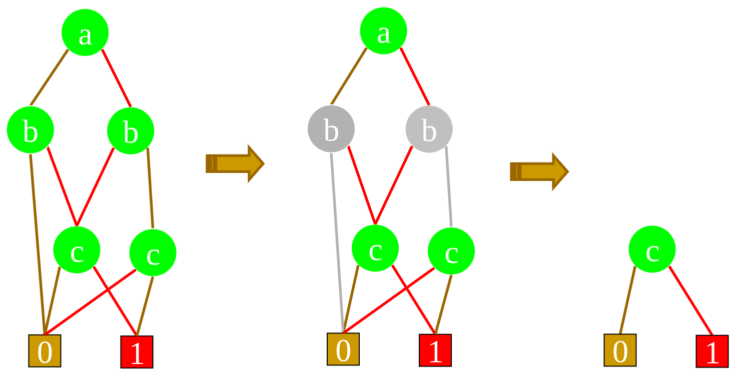
$F=ac+bc+d$ $G=ac'+d$ $F+G=a+bc+d$



Pseudocode for RESTRICT operator

```
function Restrict( F, var, value )
if ( AlreadyComputed( F, var, value ) ) return result;
else if ( Var( F ) > var ) return F;
else if ( Var( F ) < var )
    u = CreateNode( Var(F), Restrict(Fx′, var, value),
                              Restrict(Fx,  var, value) );
    InsertComputed(F, var, value, u ); return u;
else /* ( Var( F ) == var */ if ( value == 0 )
     return = Restrict(Fx′, var, value);
else /* ( Var( F ) == var && value == 1 ) */
     return = Restrict(Fx , var, value);
```

$F = bc + ab'c'$ $F(b=1) = ?$



Derived Operations: COMPOSE

- Given F(x) and G(y), find F(G(y))
- Using Shannon Expansion

  F(x) = x′ & Fx′ + x & Fx

  F(G(y)) = G′(y) & Fx′ + G(y) & Fx
- COMPOSE is reduced to two operations RESTRICT and three operations APPLY

Derived Operations: Quantification

- Given a function $F(x_1, x_2, x_3)$
- Existential quantification of F w.r.t. x_1 is
  $\exists_{x1} F(x_1, x_2, x_3) = F(0, x_2, x_3) + F(1, x_2, x_3)$
- Universal quantification of F w.r.t. x_1 is
  $\forall_{x1} F(x_1, x_2, x_3) = F(0, x_2, x_3) \& F(1, x_2, x_3)$
- Unique quantification of F w.r.t. x_1 is
  $!_{x1} F(x_1, x_2, x_3) = F(0, x_2, x_3) \oplus F(1, x_2, x_3)$

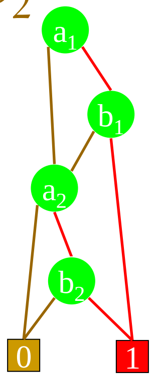
Summery of Operations on BDDs

- Apply – NOT, AND, OR, EXOR, etc.
- Restrict
- Compose (Replace)
- Quantification (existential, universal)
- Specialized operators
  - Generalized cofactor (constrain, restrict)
  - Compatible projection, etc.

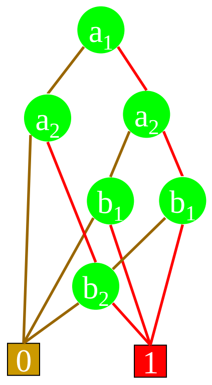
Worst-Case Complexity

- CreateNode - O(1)
- Build - O(2^n)
- APPLY - O(|F|*|G|)
- RESTRICT - O(|F|)
- COMPOSE - O($|F|^2$ *$|G|^2$)

Variable Ordering for $F = a_1b_1 + a_2b_2$

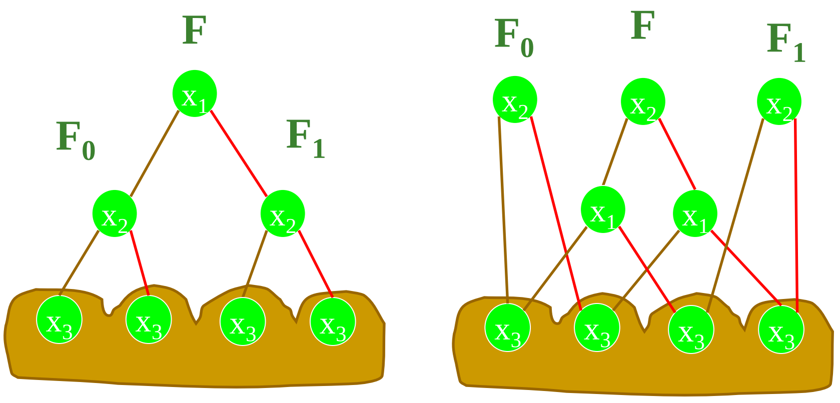


$a_1 < b_1 < a_2 < b_2$



$a_1 < a_2 < b_1 < b_2$

Dynamic Variable Reordering



Variable reordering is a local operation

Introduction to BDDs: References

- Henrik Reif Andersen. An Introduction to Binary Decision Diagrams. Dept. of Information Technology, Technical University of Denmark, 1997.

  http://andrea.it.dtu.dk/~hra/notes-index.html
- R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, Vol. 24, No. 3 (September, 1992), pp. 293-318.
- R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, Vol. C-35, No. 8 (August, 1986), pp. 677-691.

  Reprinted in M. Yoeli, *Formal Verification of Hardware Design*, IEEE Computer Society Press, 1990, pp. 253-267.

Existential Quantification

- Existential quantification (abstraction)

$$\exists_x f = f|_{x=0} + f|_{x=1}$$

- Example:

$$\exists_x (x\,y + z) = y + z$$

- <u>Note</u>**:** $\exists_x f$ does not depend on *x* (smoothing)

- Useful in symbolic image computation (*sets* of states)

Existential Quantification - cont'd

- Function can be existentially quantified w.r.to a vector: $X = x_1 x_2 \ldots$

$$\exists_X f = \exists_{x1x2\ldots} f = \exists_{x1} \exists_{x2} \exists_{\ldots} f$$

- Can be done efficiently directly on a BDD
- Very useful in computing *sets* of states
  - Image computation: *next* states
  - Pre-Image computation: *previous* states

  from a given *set* of initial states

Summary

- What is verification?
- What is logic? (completeness, soundness)
- Propositional Logic (syntax, semantics, axiom system, natural deduction, next class: sequent calculus)
- BDD