CS181u Applied Logic & Automated Reasoning

Lecture 4

Shannon Expansion
Adequate Sets of Connectives
Unit Propagation
DPLL
Binary Decision Diagrams

Variable Substitution (Replacement)

Variable replacement: given a Boolean formula f, a variable v, and an expression e, the notation $f[e/v]$ denotes the replacement of v with e in f.

Example: if $f = \neg x \wedge \neg y$ then

$$f[F/y] = \neg x \wedge \neg F = \neg x \wedge T = \neg x$$
$$f[T/x] = \neg T \wedge \neg y = F \wedge \neg y = F$$
$$f[\neg r/y] = \neg x \wedge \neg\neg r = \neg x \wedge r$$

Shannon Expansion

Given a variable x in a formula f, x can either be true or false. So, we can split the formula into two pieces—one in which x is asserted to be false and we plug F into f for x, and one in which x is aserted to be true and we plug T into f for x—and then take the disjunction.

Shannon Expansion

Given a variable x in a formula f, x can either be true or false. So, we can split the formula into two pieces—one in which x is asserted to be false and we plug F into f for x, and one in which x is aserted to be true and we plug T into f for x—and then take the disjunction.

Hence, we can write the logical equivalence:

$$f \equiv (x = F) \wedge f[F/x] \quad \vee \quad (x = T) \wedge f[T/x]$$

Shannon Expansion

Given a variable x in a formula f, x can either be true or false. So, we can split the formula into two pieces—one in which x is asserted to be false and we plug F into f for x, and one in which x is aserted to be true and we plug T into f for x—and then take the disjunction.

Hence, we can write the logical equivalence:

$$f \equiv (x = F) \wedge f[F/x] \quad \vee \quad (x = T) \wedge f[T/x]$$

Or, equivalently

$$f \equiv (\neg x) \wedge f[F/x] \quad \vee \quad x \wedge f[T/x]$$

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Proposition: the set $S = \{\neg, \land, \lor\}$ is adequate.

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Proposition: the set $S = \{\neg, \wedge, \vee\}$ is adequate.

Idea: induction on the number of variables in formula f

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Proposition: the set $S = \{\neg, \wedge, \vee\}$ is adequate.

Idea: induction on the number of variables in formula f

Base Case: f has 0 variables

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Proposition: the set $S = \{\neg, \wedge, \vee\}$ is adequate.

Idea: induction on the number of variables in formula f

Base Case: f has 0 variables

Then f is equivalant to T or F, both of which do not use connectives outside of S.

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Proposition: the set $S = \{\neg, \wedge, \vee\}$ is adequate.

Idea: induction on the number of variables in formula f

Base Case: f has 0 variables

Then f is equivalant to T or F, both of which do not use connectives outside of S.

Inductive Step: f has n variables: $x_1, \ldots, x_n$

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Proposition: the set $S = \{\neg, \wedge, \vee\}$ is adequate.

Idea: induction on the number of variables in formula f

Base Case: f has 0 variables

Then f is equivalant to T or F, both of which do not use connectives outside of S.

Inductive Step: f has n variables: $x_1, \ldots, x_n$

Assume all formulas with less than n variables can be written using only S connectives.

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Proposition: the set $S = \{\neg, \wedge, \vee\}$ is adequate.

Idea: induction on the number of variables in formula f

Base Case: f has 0 variables

Then f is equivalant to T or F, both of which do not use connectives outside of S.

Inductive Step: f has n variables: $x_1, \ldots, x_n$

Assume all formulas with less than n variables can be written using only S connectives.

Shannon expand: $f \equiv \neg x \wedge f[F/x] \quad \vee \quad x \wedge f[T/x]$

Adequate Set of Connectives

A set of connectives, S, is called adequate iff every truth function can be written equivalently using connectives from S.

Proposition: the set $S = \{\neg, \wedge, \vee\}$ is adequate.

Idea: induction on the number of variables in formula f

Base Case: f has 0 variables

Then f is equivalant to T or F, both of which do not use connectives outside of S.

Inductive Step: f has n variables: $x_1, \ldots, x_n$

Assume all formulas with less than n variables can be written using only S connectives.

Shannon expand: $f \equiv \neg x \wedge f[F/x] \quad \vee \quad x \wedge f[T/x]$

$f[F/x_1]$ and $f[T/x_1]$ both have less than n variables and so can be written with just $\{\neg, \wedge, \vee\}$.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, *u*.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad \neg x \vee z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad \neg x \vee z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \quad \neg x \lor z, z \lor w, x, y \lor v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad\qquad\qquad z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad\qquad z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad\qquad z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.
3. repeat.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad\qquad\qquad z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.
3. repeat.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad\qquad z, z \vee w, x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.
3. repeat.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad\qquad z, \qquad x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.
3. repeat.

Boolean Satisfiability

DPLL uses **Unit Propagation**.

$$\phi = \{ \qquad z, \qquad x, y \vee v\}$$

A *unit clause* is a clause that is composed of a single literal, u.

1. remove every clause (other than the unit clause) containing u.
2. in every clause that contains $\neg u$, delete $\neg u$.
3. repeat.

$\phi = \{z, x, y \vee v\}$

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : CNF formula ϕ over *n* variables
Output : true or false, the satisfiability of F
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** false
  if all clauses of ϕ satisfied **then return** true
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto \mathit{true}]$) $\vee$ DPLL($\phi[x \mapsto \mathit{false}]$)
end

DPLL Execution Example

$$\{z, x, y \vee v\}$$

DPLL Execution Example

$\{z, x, y \vee v\}$

$x \mapsto F$

UNSAT $\{z, F, y \vee v\}$

DPLL Execution Example

$$\begin{tikzcd}
& & \{z, x, y \vee v\} \arrow[dl, dash] \arrow[dr, dash] & \\
& x \mapsto F \arrow[dl, dash] & & x \mapsto T \arrow[d, dash] \\
\text{UNSAT} \quad \{z, F, y \vee v\} & & & \{z, T, y \vee v\}
\end{tikzcd}$$

DPLL Execution Example

$\{z, x, y \vee v\}$

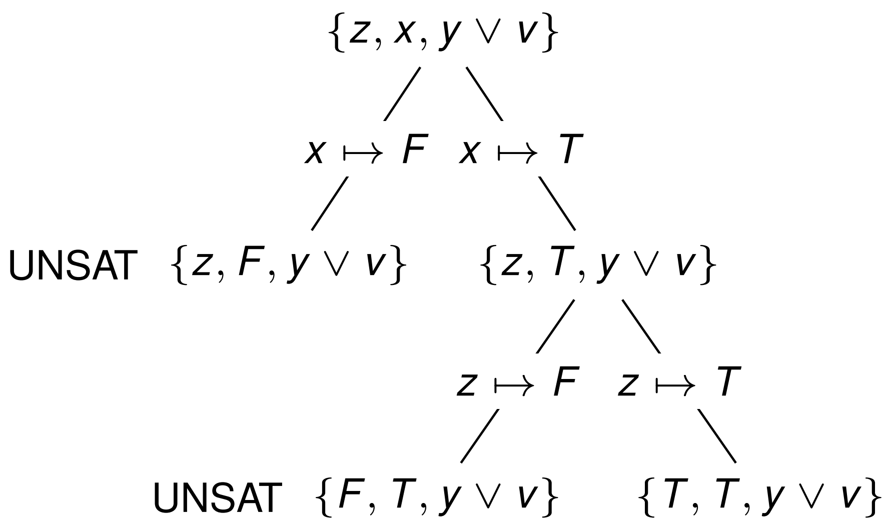
$x \mapsto F$ $\quad$ $x \mapsto T$

UNSAT $\{z, F, y \vee v\}$ $\quad$ $\{z, T, y \vee v\}$

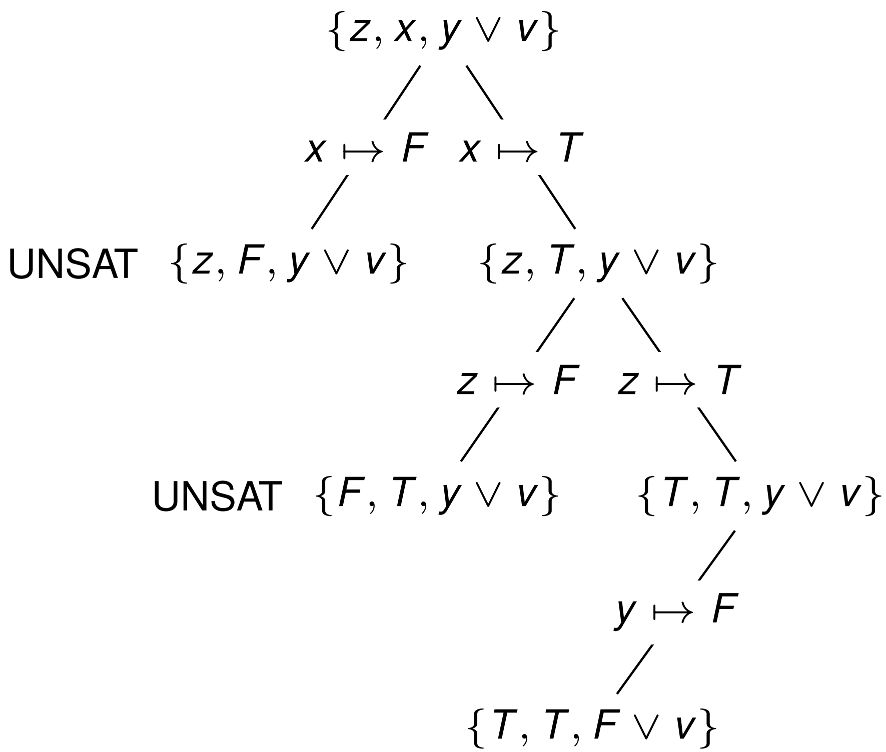
$z \mapsto F$

UNSAT $\{F, T, y \vee v\}$

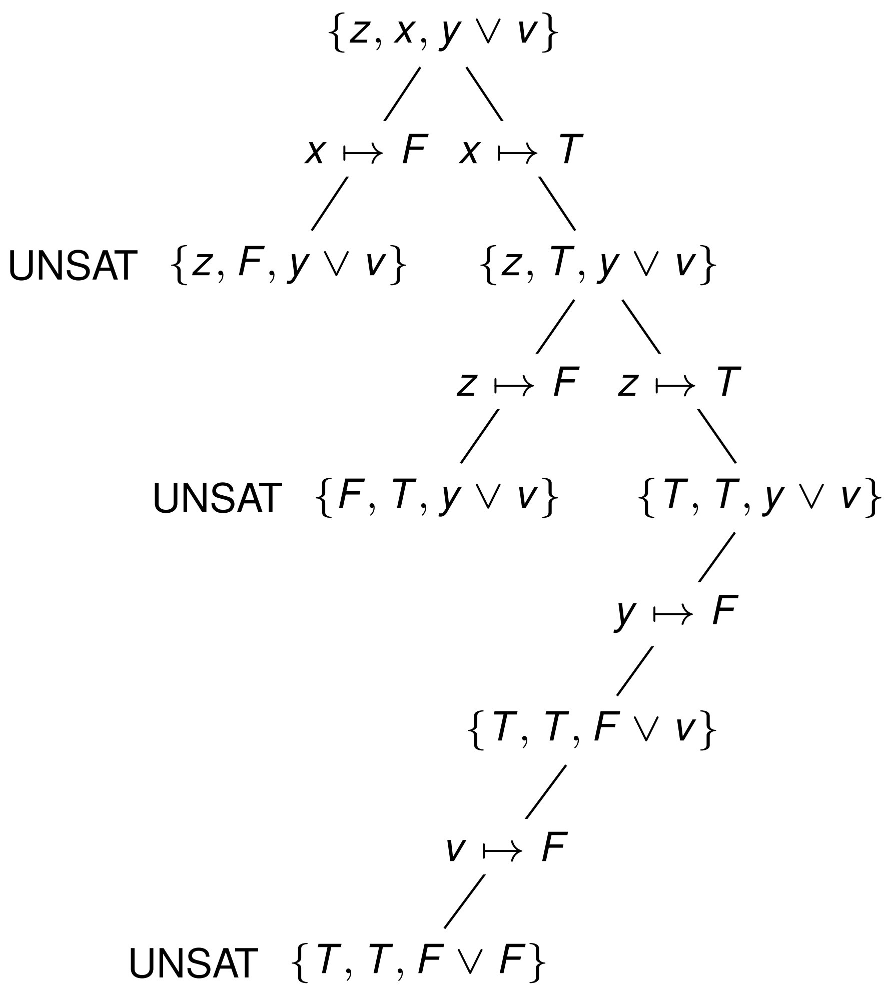
DPLL Execution Example



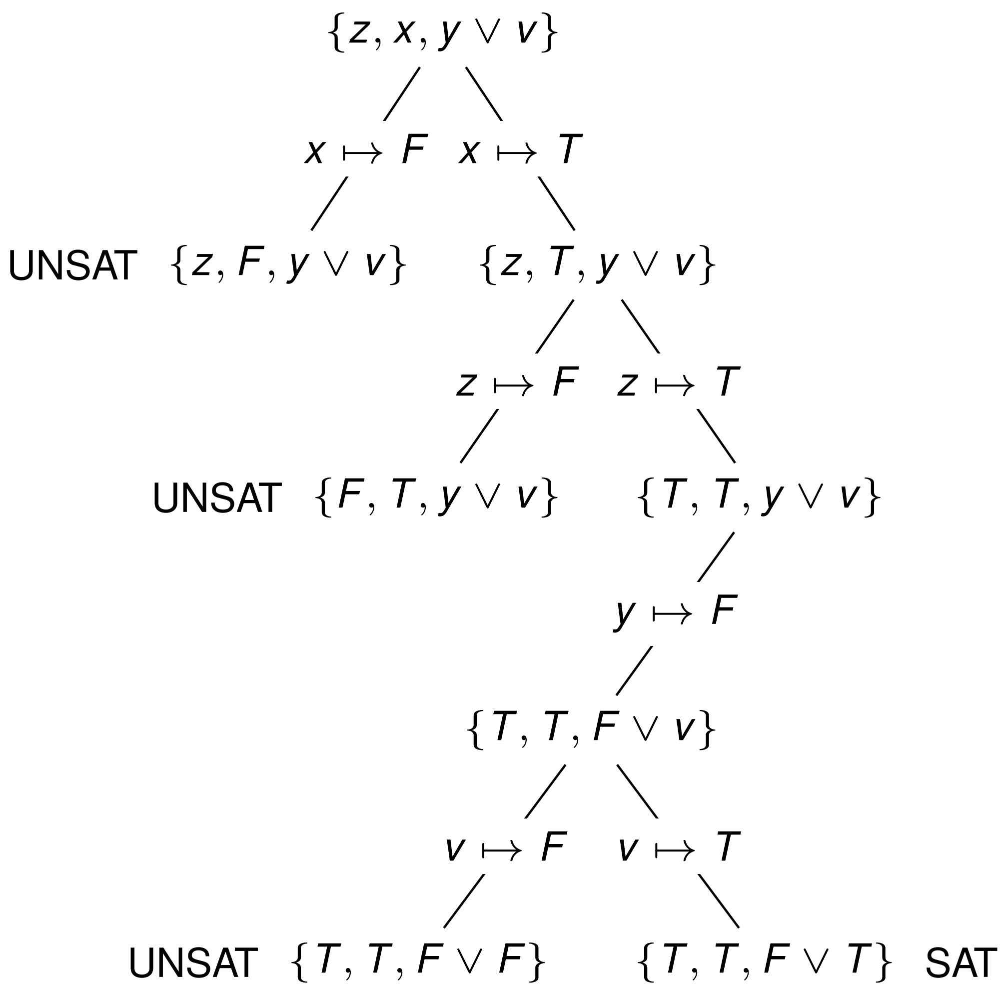
DPLL Execution Example



DPLL Execution Example



DPLL Execution Example



DPLL Execution Example

$$\{z, x, y \vee v\}$$

$$x \mapsto F \qquad x \mapsto T$$

UNSAT $\{z, F, y \vee v\}$ $\qquad$ $\{z, T, y \vee v\}$

$$z \mapsto F \qquad z \mapsto T$$

UNSAT $\{F, T, y \vee v\}$ $\qquad$ $\{T, T, y \vee v\}$

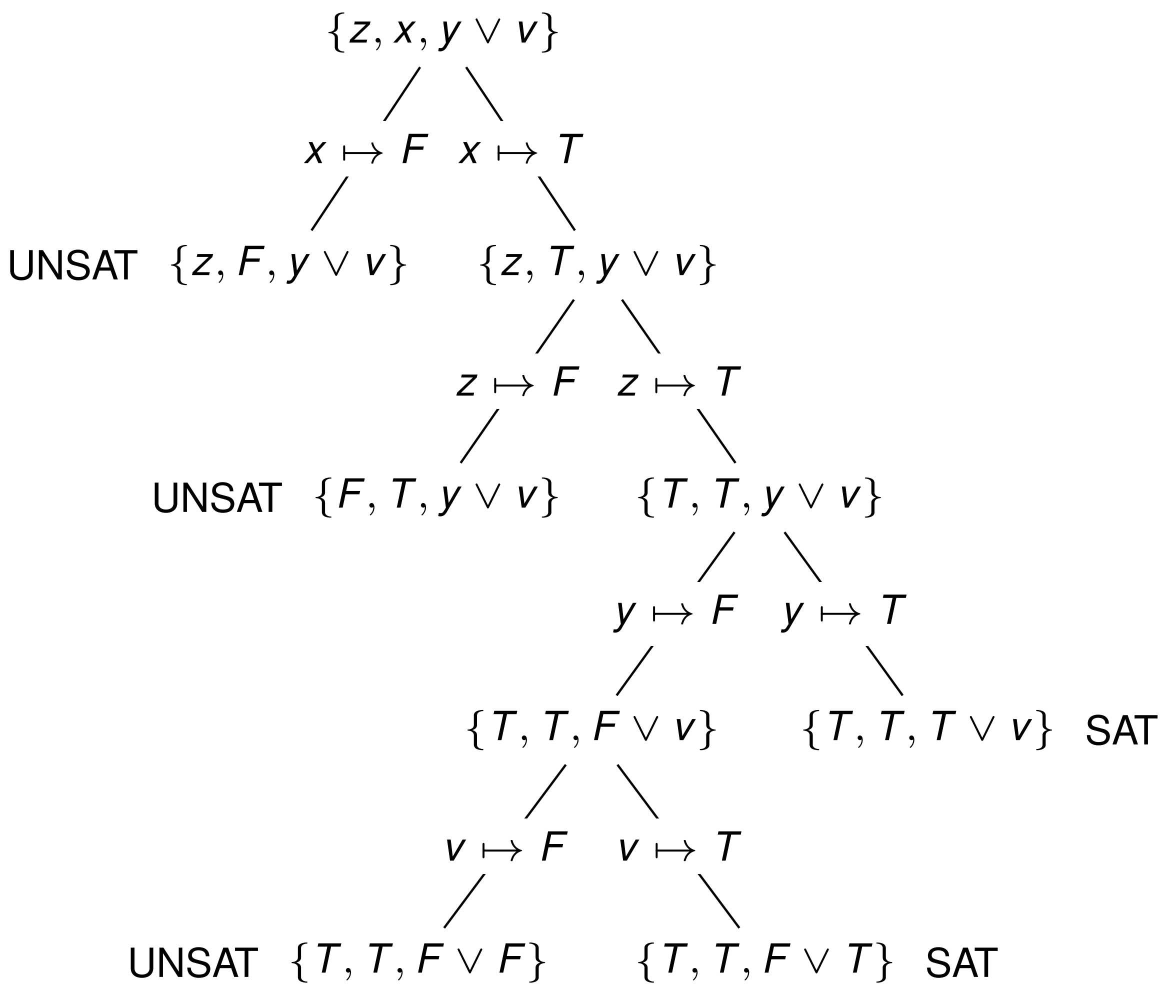
$$y \mapsto F \qquad y \mapsto T$$

$\{T, T, F \vee v\}$ $\qquad$ $\{T, T, T \vee v\}$ SAT

$$v \mapsto F \qquad v \mapsto T$$

UNSAT $\{T, T, F \vee F\}$ $\qquad$ $\{T, T, F \vee T\}$ SAT

DPLL Execution Example



Result: ϕ is satisfiable.

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

Function : DPLL(ϕ)
Input : CNF formula ϕ over *n* variables
Output : true or false, the satisfiability of F
begin

   UnitPropagate(ϕ)
   if ϕ has false clause **then return** false
   if all clauses of ϕ satisfied **then return** true
   x $\leftarrow$ SelectBranchVariable(ϕ)
   return DPLL($\phi[x \mapsto \mathit{true}]$) $\vee$ DPLL($\phi[x \mapsto \mathit{false}]$)

end

Davis-Putnam-Logemann-Loveland (DPLL) Algorithm

DPLL can be converted into a procedure for $\#$CNF-SAT.

Function : DPLL(ϕ, t)
Input : CNF formula ϕ over n variables; $t \in \mathbb{Z}$
Output : $\#\phi$, the model count of ϕ
begin
  UnitPropagate(ϕ)
  if ϕ has false clause **then return** 0
  if all clauses of ϕ satisfied **then return** 2^t
  x $\leftarrow$ SelectBranchVariable(ϕ)
  return DPLL($\phi[x \mapsto \mathit{true}], t - 1$) $+$ DPLL($\phi[x \mapsto \mathit{true}], t - 1$)
end

Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$

$\{z, x, y \vee v\}$ $t = 5$

Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$

$\{z, x, y \vee v\}$ $t = 5$

$x \mapsto F$

0 $\{z, F, y \vee v\}$ $t = 4$

Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$

$\{z, x, y \vee v\}$ $t = 5$

$x \mapsto F$ $\qquad$ $x \mapsto T$

0 $\{z, F, y \vee v\}$ $t = 4$ $\qquad$ $\{z, T, y \vee v\}$ $t = 4$

Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$

$\{z, x, y \vee v\}$ $t = 5$

$x \mapsto F$ $\qquad$ $x \mapsto T$

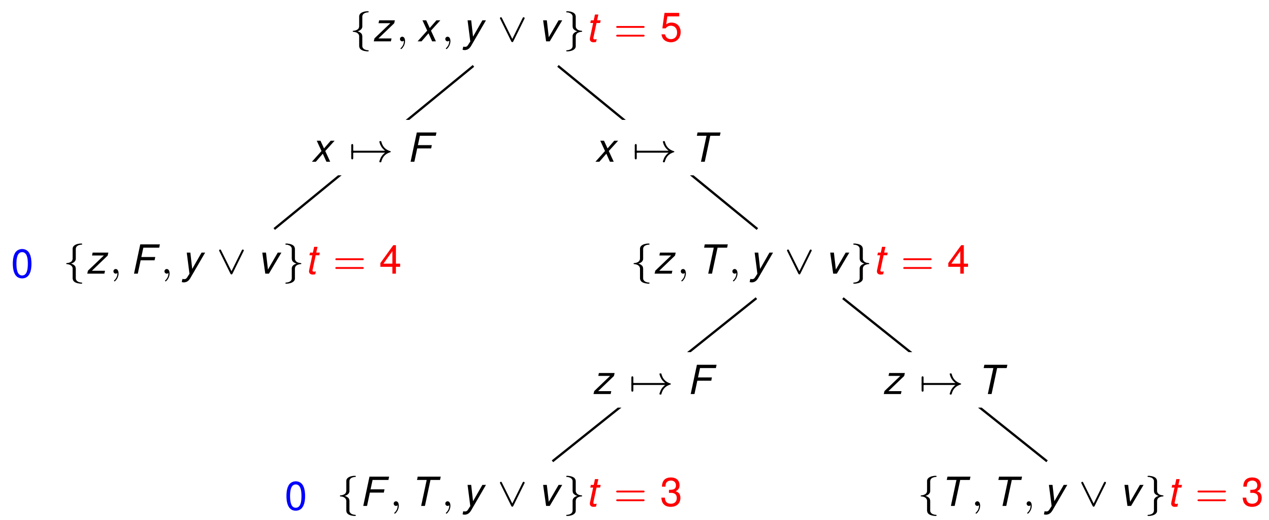
0 $\{z, F, y \vee v\}$ $t = 4$ $\qquad$ $\{z, T, y \vee v\}$ $t = 4$

$z \mapsto F$

0 $\{F, T, y \vee v\}$ $t = 3$

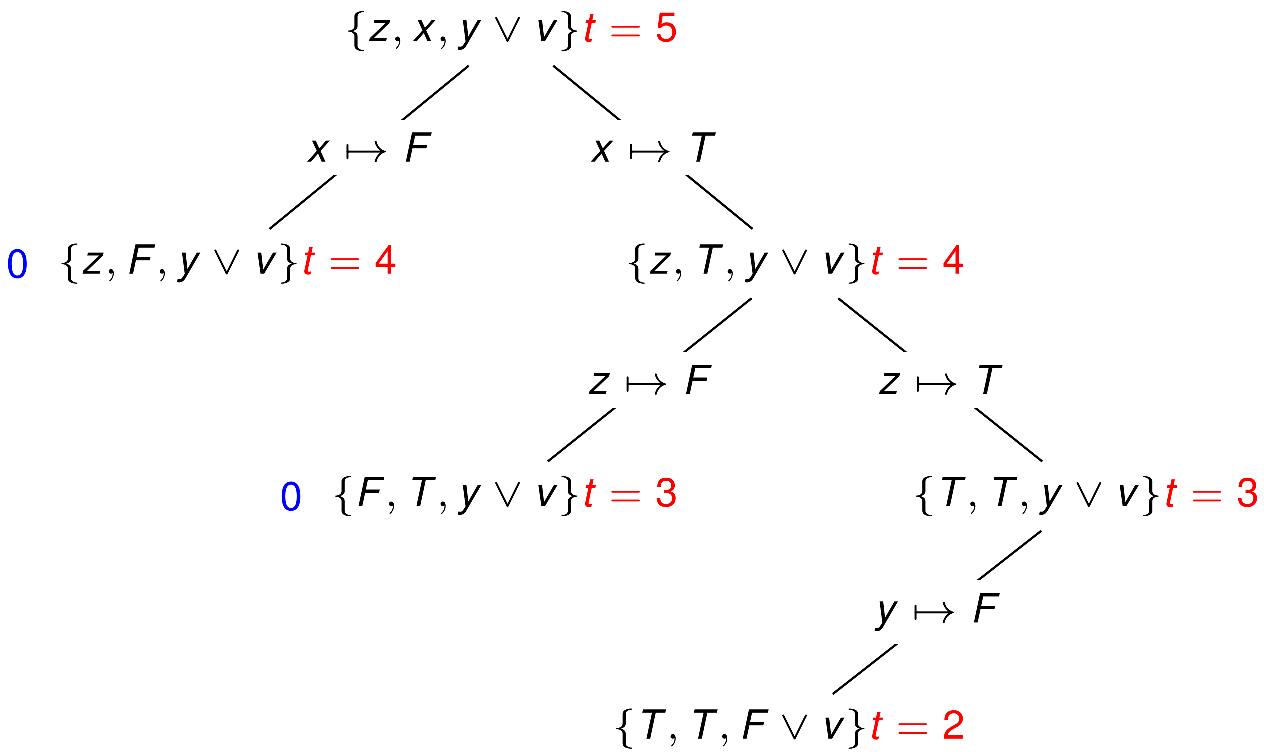
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



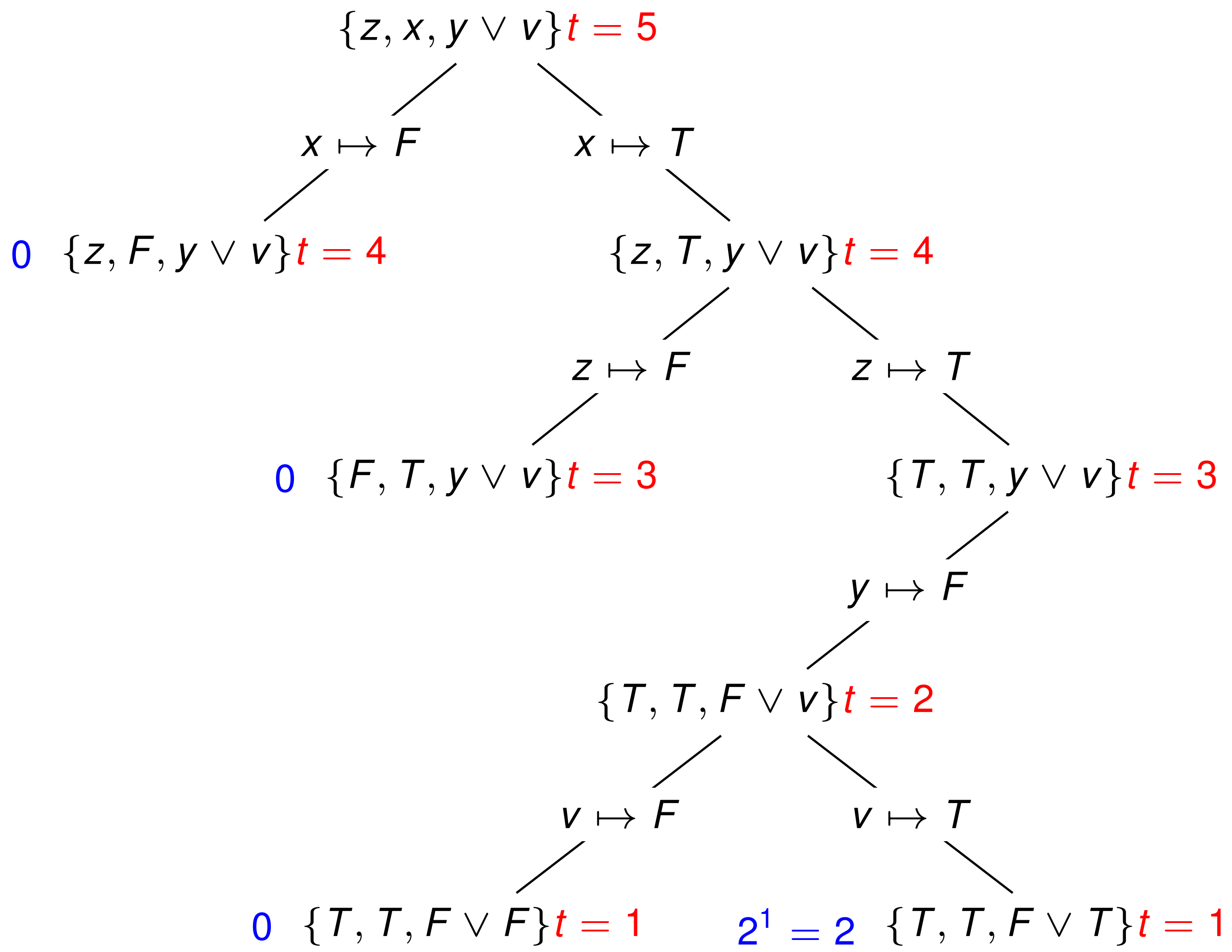
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$

- $\{z, x, y \vee v\}$ $t = 5$
  - $x \mapsto F$
    - 0 $\{z, F, y \vee v\}$ $t = 4$
  - $x \mapsto T$
    - $\{z, T, y \vee v\}$ $t = 4$
      - $z \mapsto F$
        - 0 $\{F, T, y \vee v\}$ $t = 3$
      - $z \mapsto T$
        - $\{T, T, y \vee v\}$ $t = 3$
          - $y \mapsto F$
            - $\{T, T, F \vee v\}$ $t = 2$
              - $v \mapsto F$
                - 0 $\{T, T, F \vee F\}$ $t = 1$

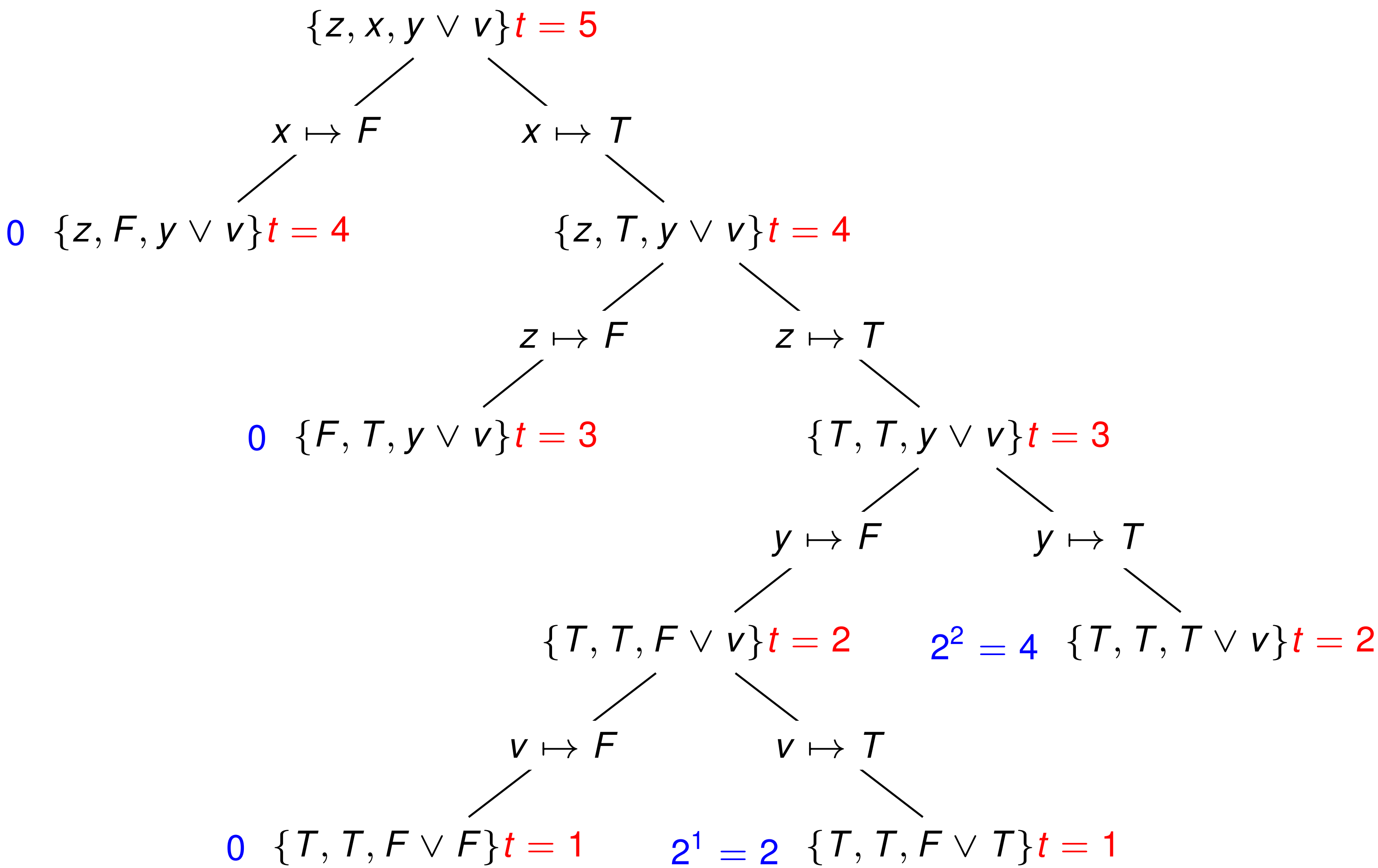
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



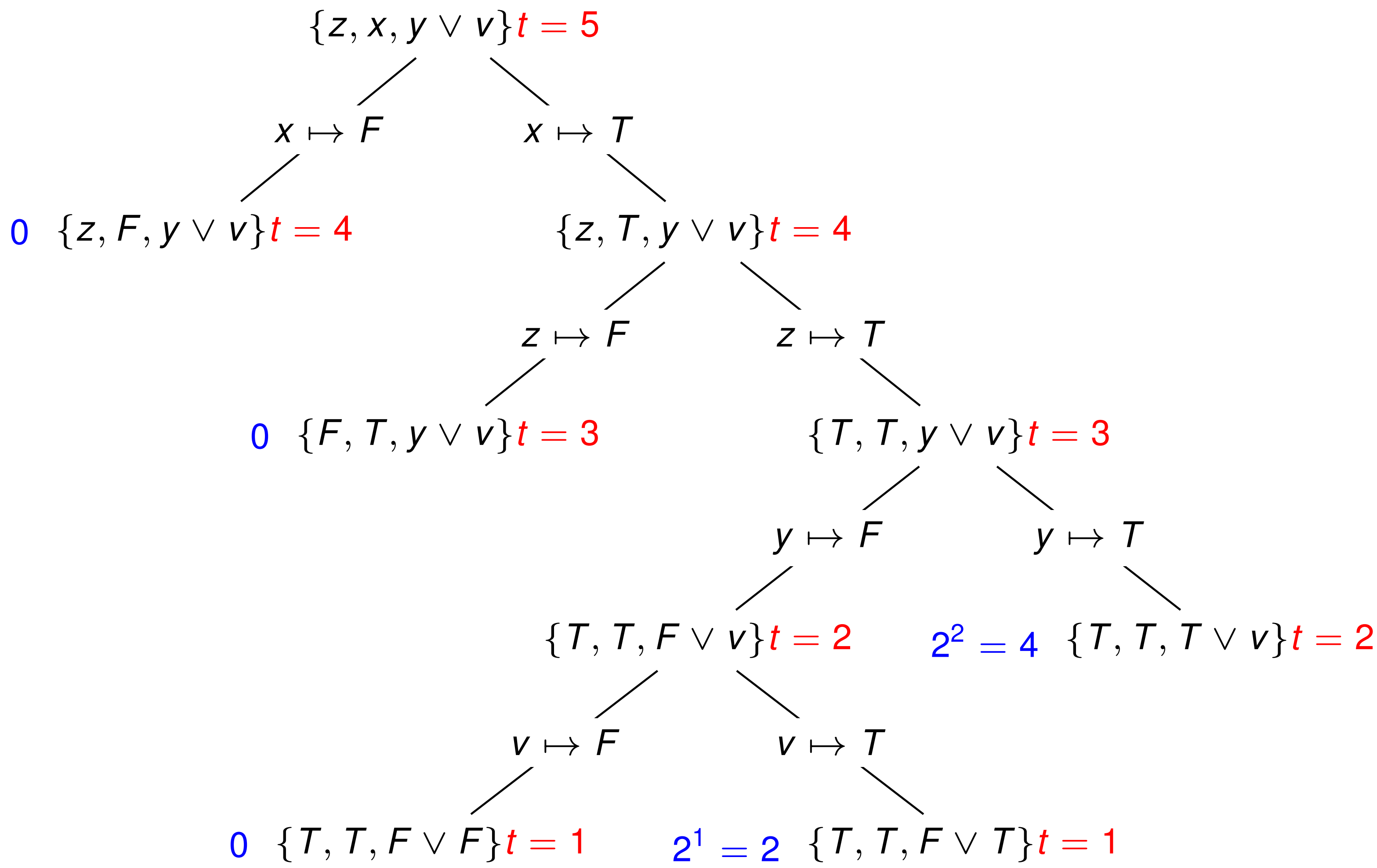
Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



Counting with DPLL

$$\phi = \{x \vee y, \neg x \vee z, z \vee w, x, y \vee v\}, n = 5$$



Result: $0 + 0 + 0 + 2 + 4 = 6$ models

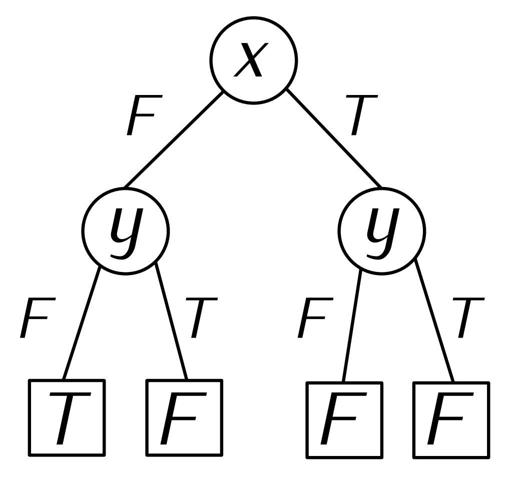
Binary Decision Diagrams

A **Binary Decision Diagram (BDD)** is a data structure for representing the truth values of formulas in propositional logic.

Example: consider the formula $\neg x \wedge \neg y$ and the truth table.

x	y	$\neg x \wedge \neg y$
F	F	T
F	T	F
T	F	F
T	T	F

The same information can be encoded in a decision tree.



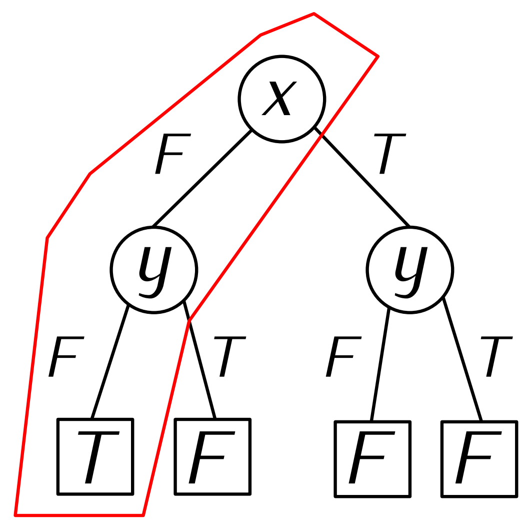
Binary Decision Diagrams

A **Binary Decision Diagram (BDD)** is a data structure for representing the truth values of formulas in propositional logic.

Example: consider the formula $\neg x \wedge \neg y$ and the truth table.

x	y	$\neg x \wedge \neg y$
F	F	T
F	T	F
T	F	F
T	T	F

The same information can be encoded in a decision tree.



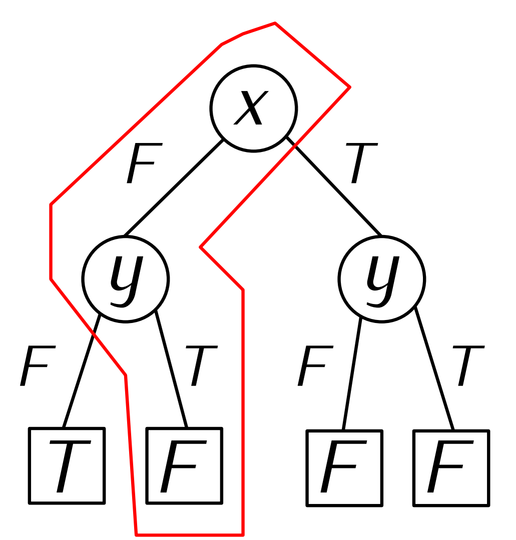
Binary Decision Diagrams

A **Binary Decision Diagram (BDD)** is a data structure for representing the truth values of formulas in propositional logic.

Example: consider the formula $\neg x \wedge \neg y$ and the truth table.

x	y	$\neg x \wedge \neg y$
F	F	T
F	T	F
T	F	F
T	T	F

The same information can be encoded in a decision tree.



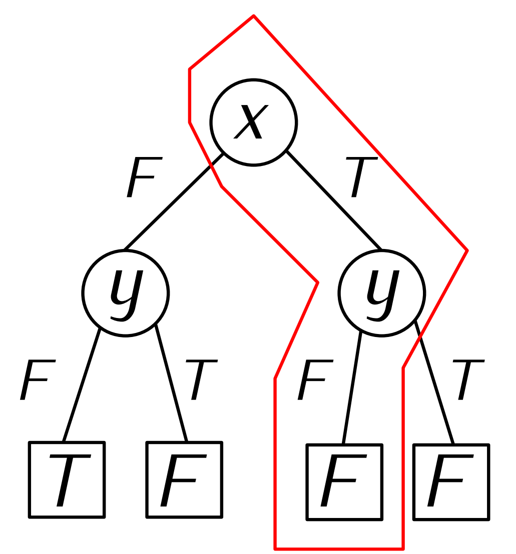
Binary Decision Diagrams

A **Binary Decision Diagram (BDD)** is a data structure for representing the truth values of formulas in propositional logic.

Example: consider the formula $\neg x \wedge \neg y$ and the truth table.

x	y	$\neg x \wedge \neg y$
F	F	T
F	T	F
T	F	F
T	T	F

The same information can be encoded in a decision tree.



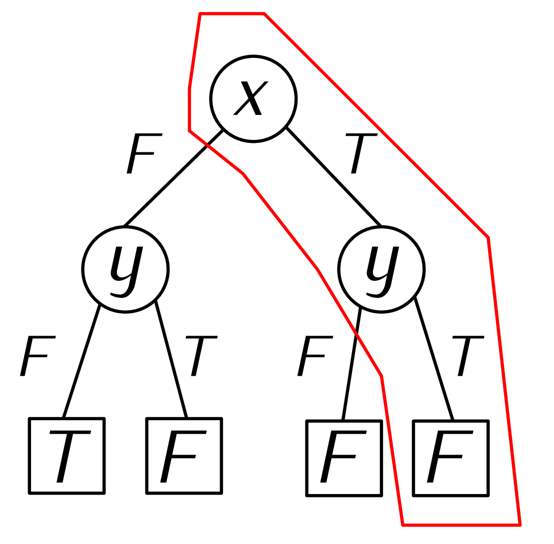
Binary Decision Diagrams

A **Binary Decision Diagram (BDD)** is a data structure for representing the truth values of formulas in propositional logic.

Example: consider the formula $\neg x \wedge \neg y$ and the truth table.

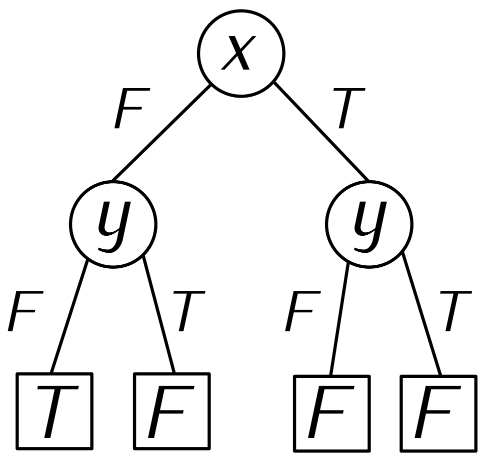
x	y	$\neg x \wedge \neg y$
F	F	T
F	T	F
T	F	F
T	T	F

The same information can be encoded in a decision tree.



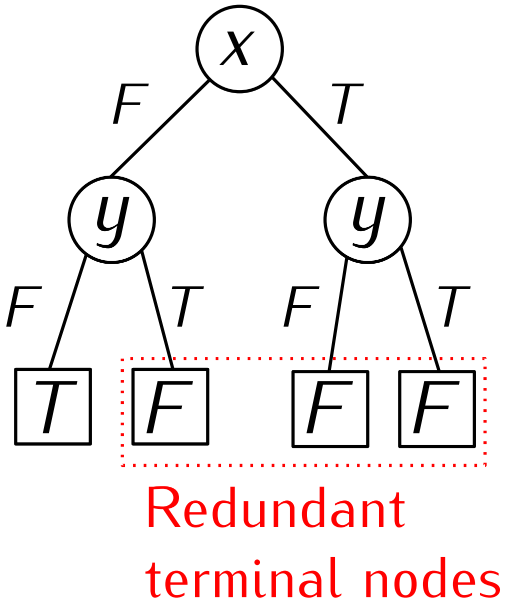
Binary Decision Diagrams

We can make the decision diagram more compact.



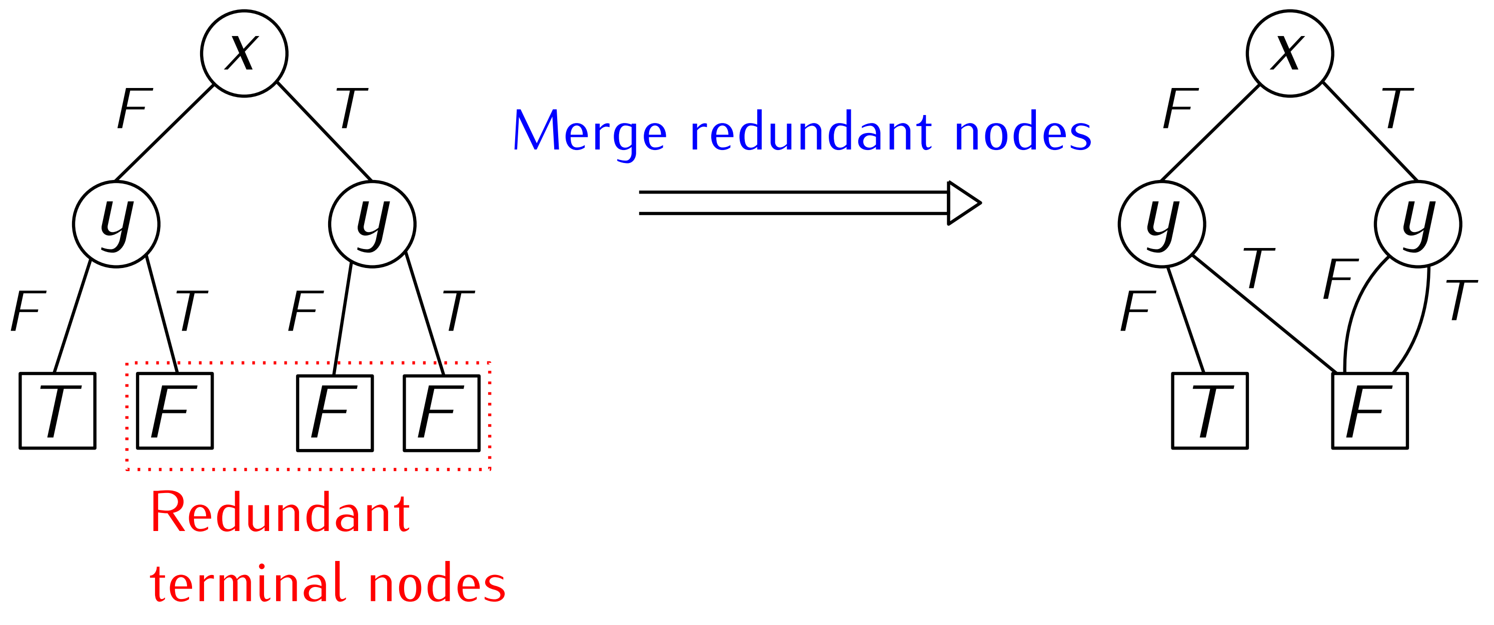
Binary Decision Diagrams

We can make the decision diagram more compact.



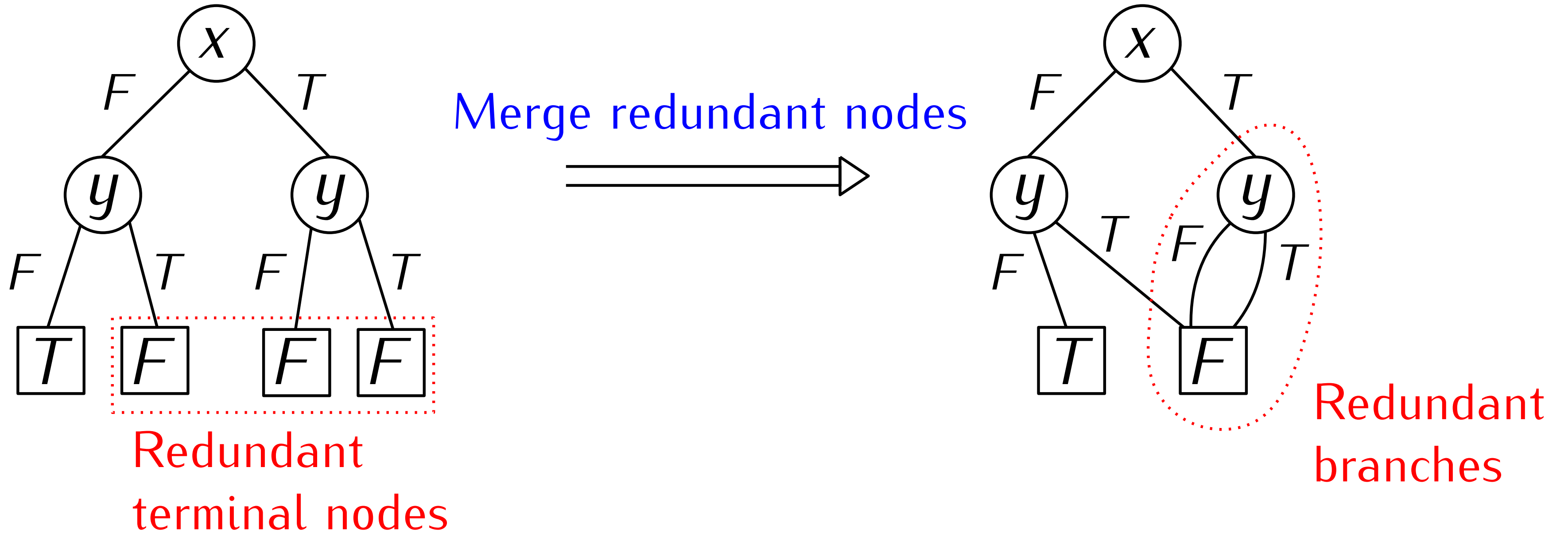
Binary Decision Diagrams

We can make the decision diagram more compact.



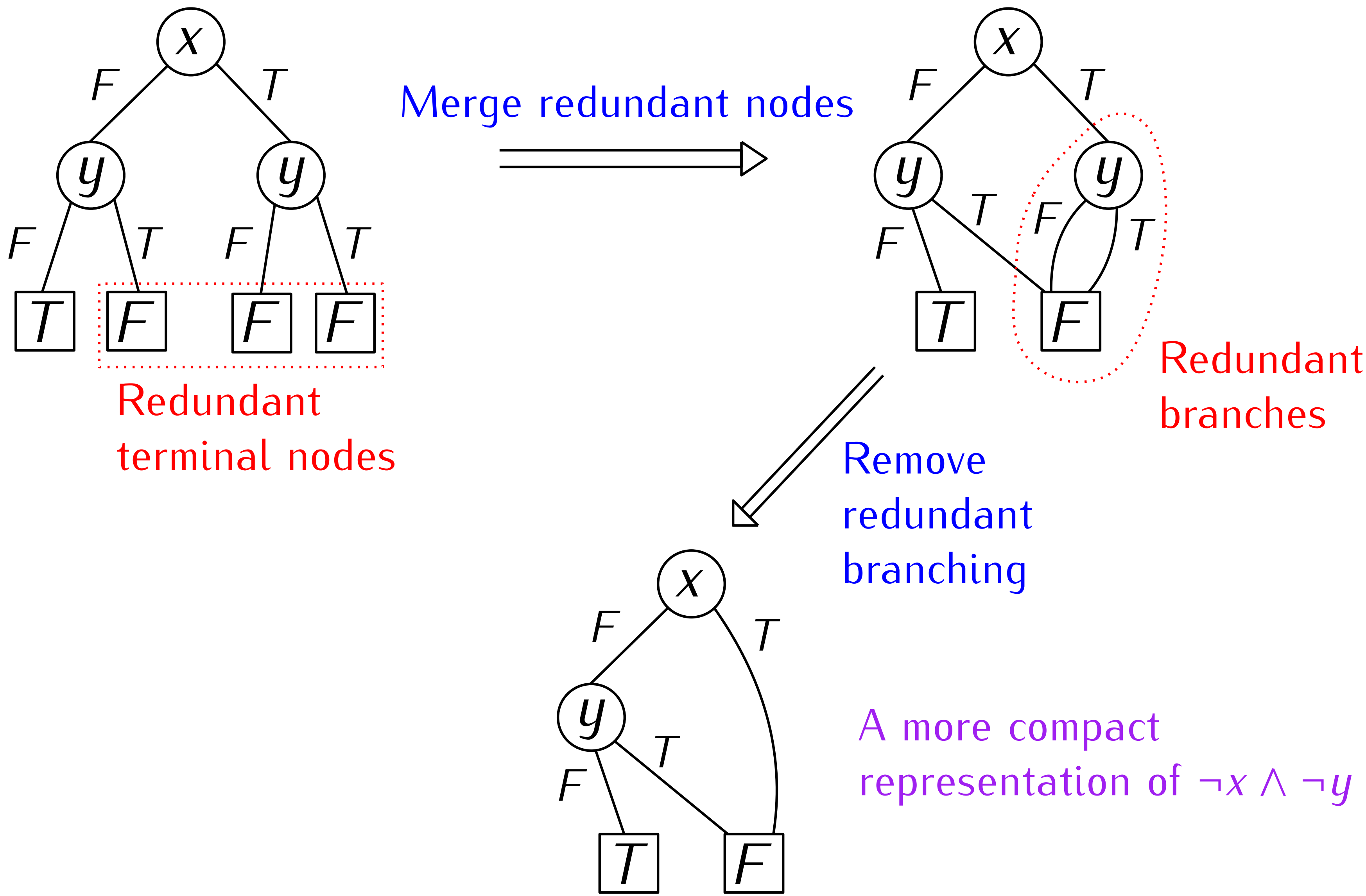
Binary Decision Diagrams

We can make the decision diagram more compact.

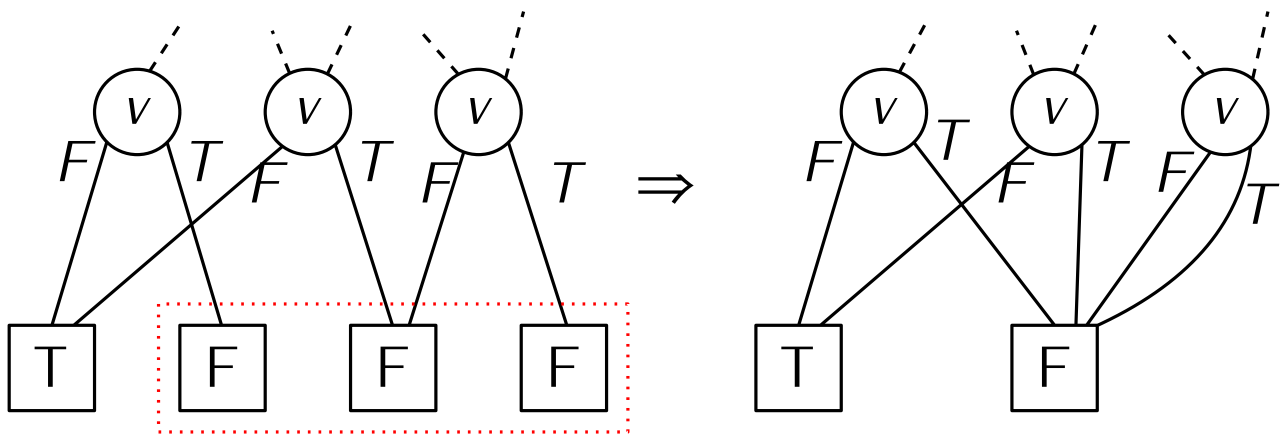


Binary Decision Diagrams

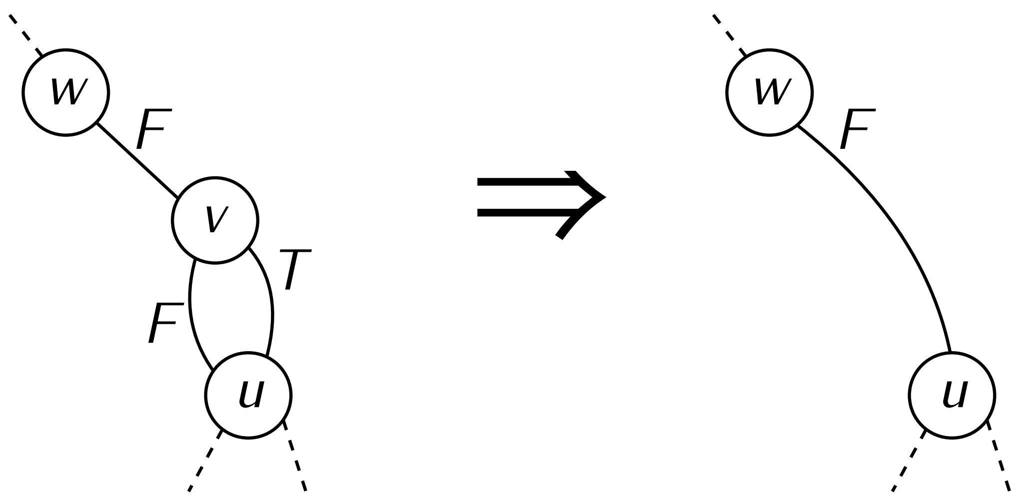
We can make the decision diagram more compact.



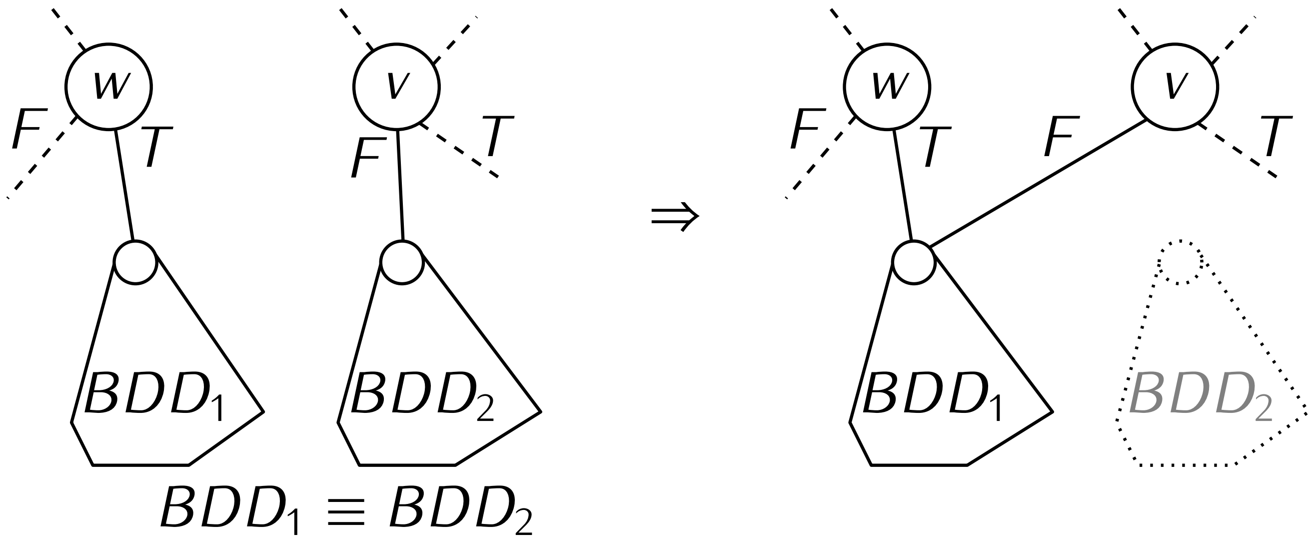
Reduction Rule 1: Merge duplicated terminal nodes.



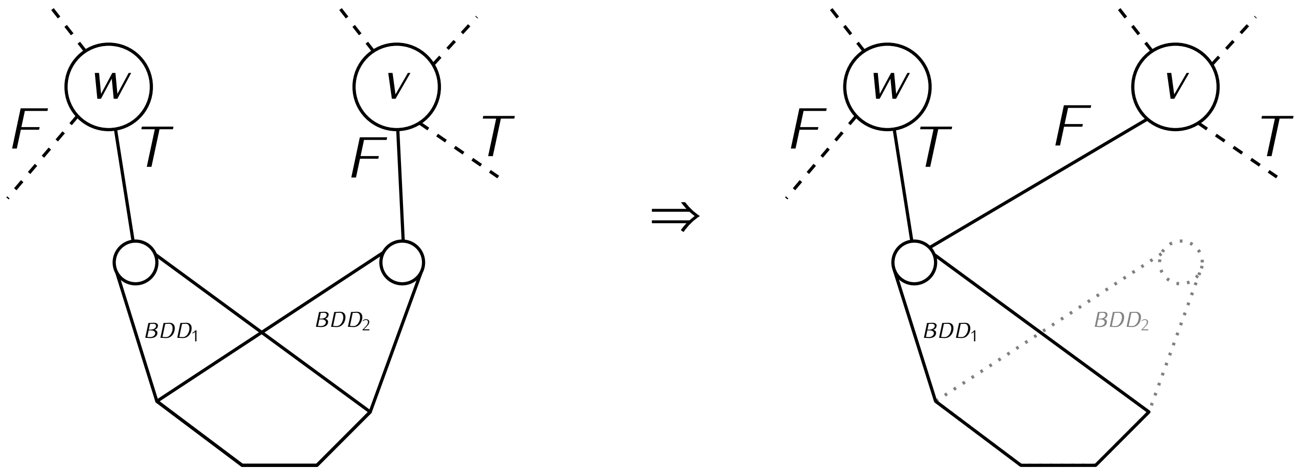
Reduction Rule 2: Remove redundant tests.



Reduction Rule 3: Remove duplicate sub-BDDs.



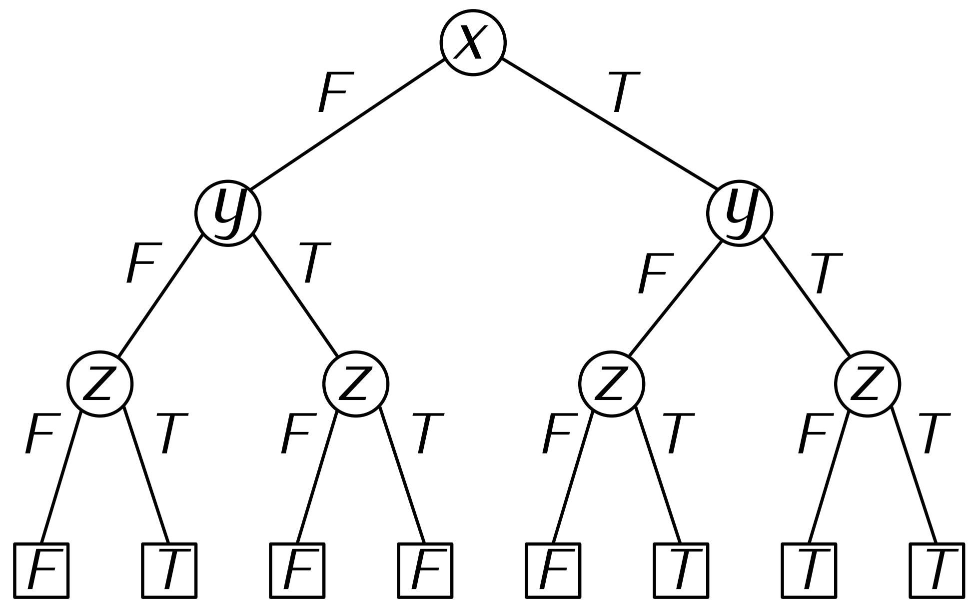
Reduction Rule 3: Remove duplicate sub-BDDs.



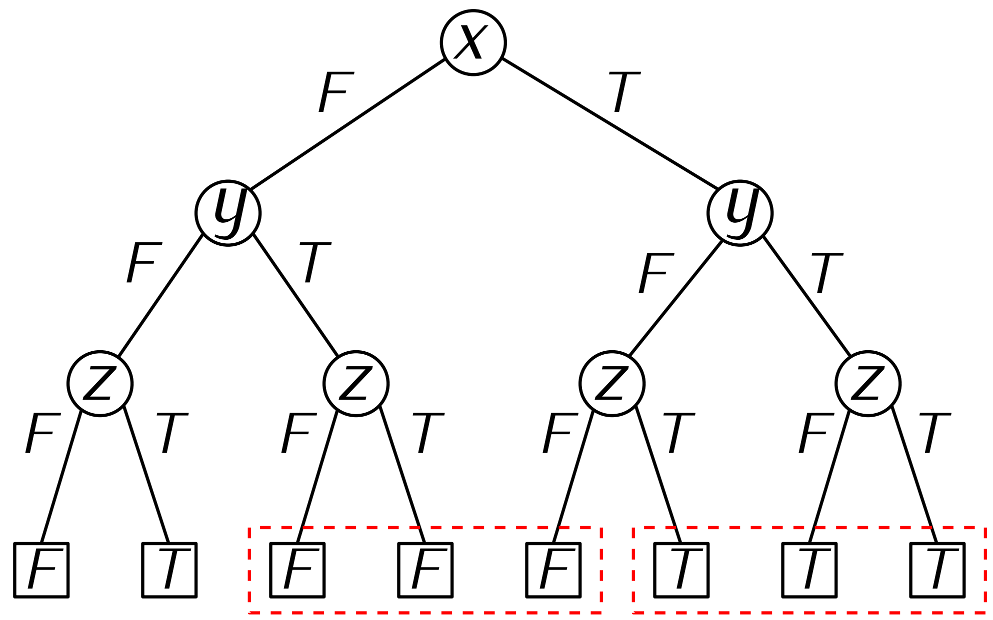
$BDD_1 \equiv BDD_2$

NOTE: They can be structurally identical, even if they overlap.

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.

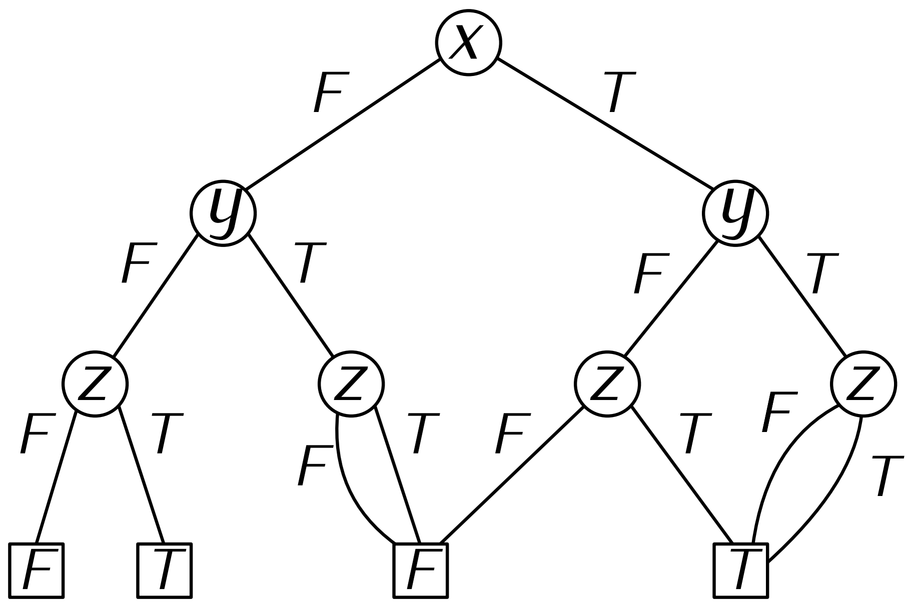


Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



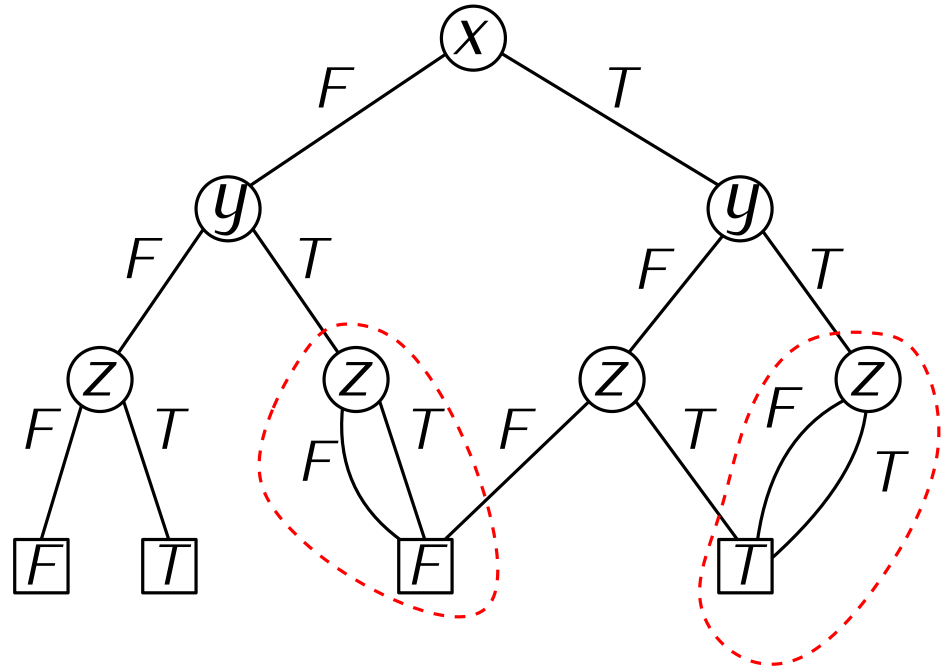
Merge duplicate terminals

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



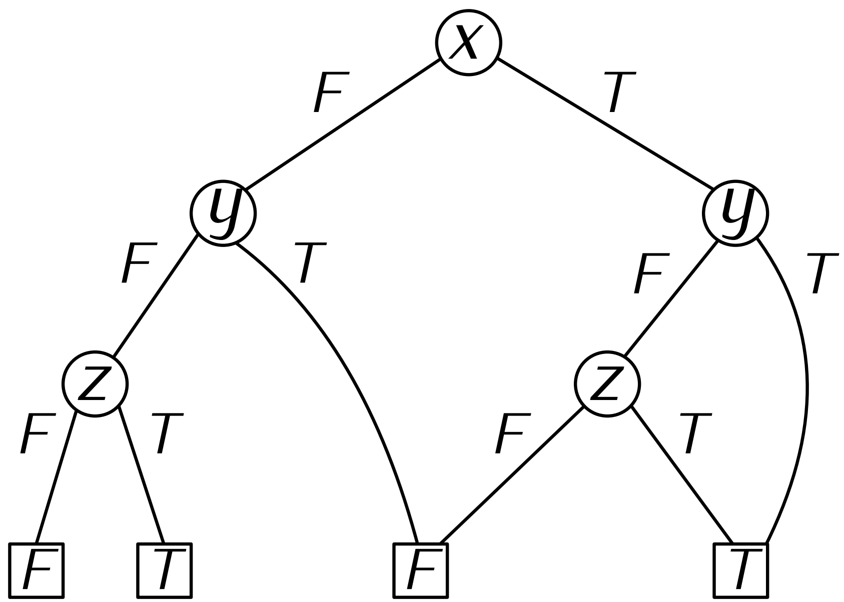
Merge duplicate terminals

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



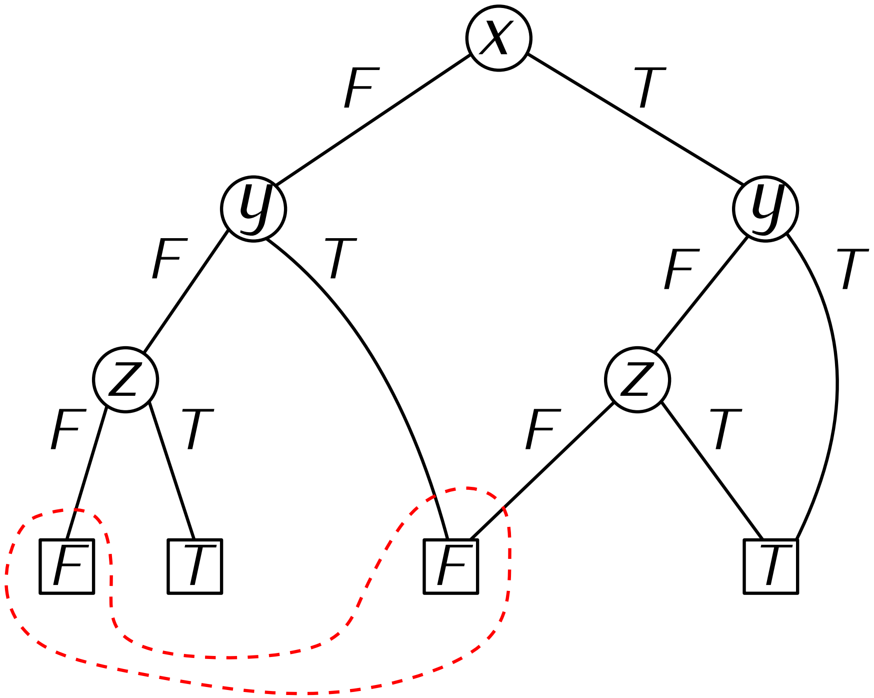
Delete redundant tests

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



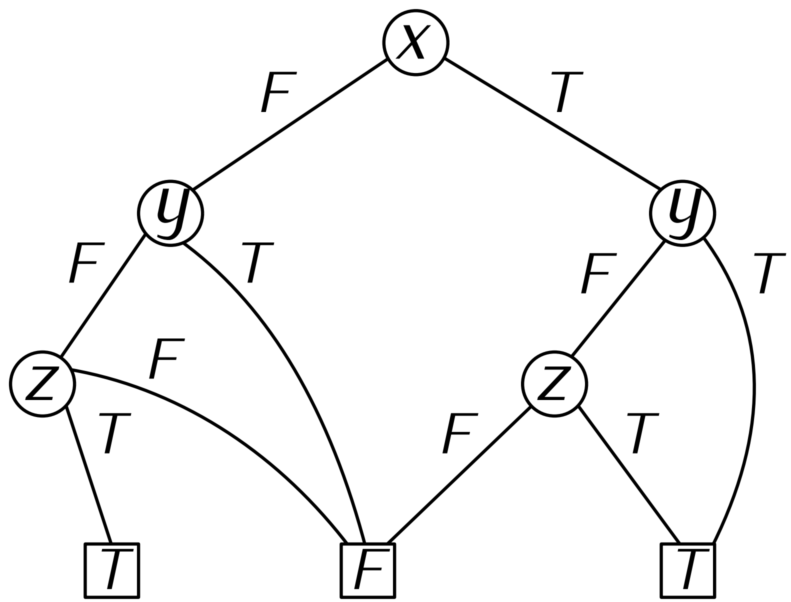
Delete redundant tests

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



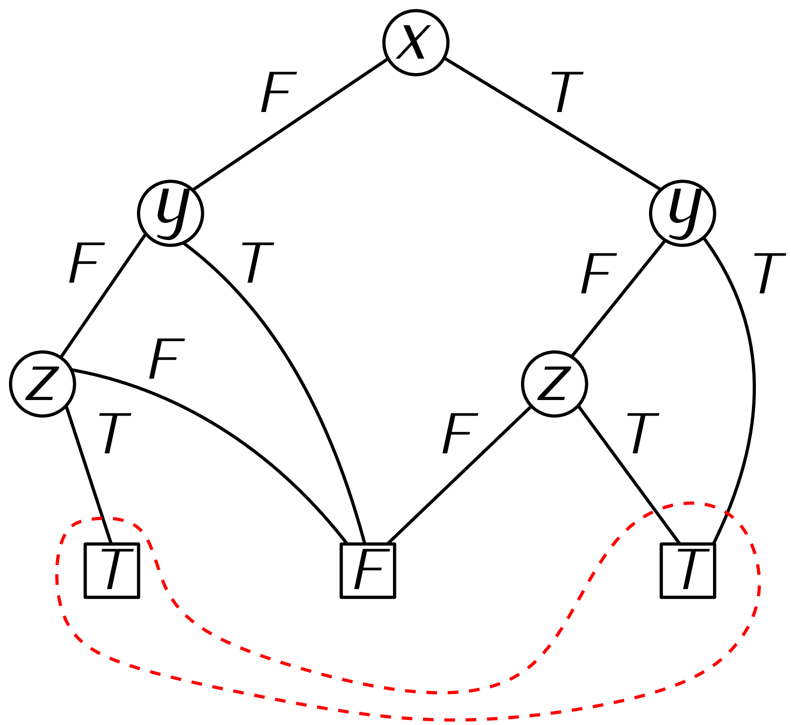
Merge more duplicate terminals

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



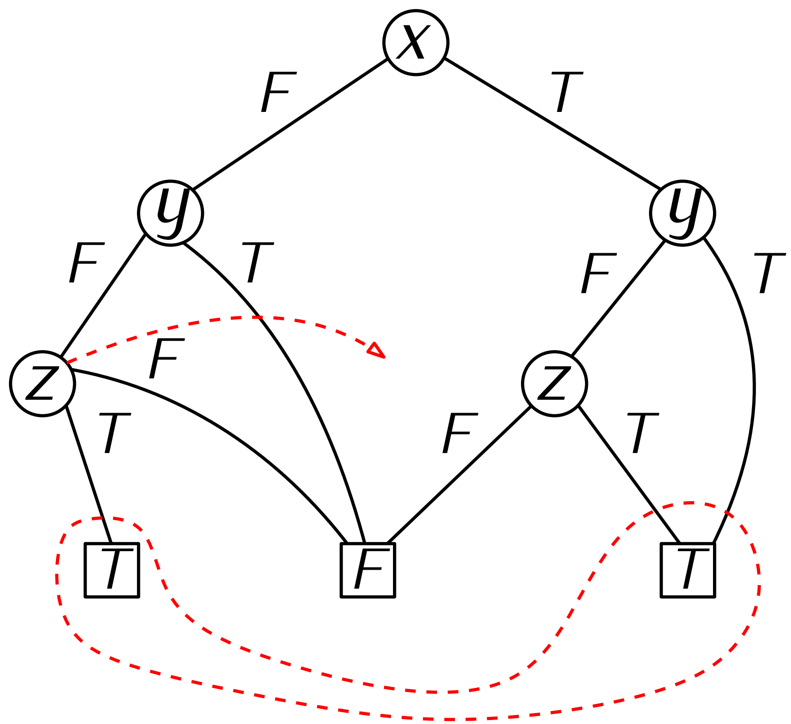
Merge more duplicate terminals

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



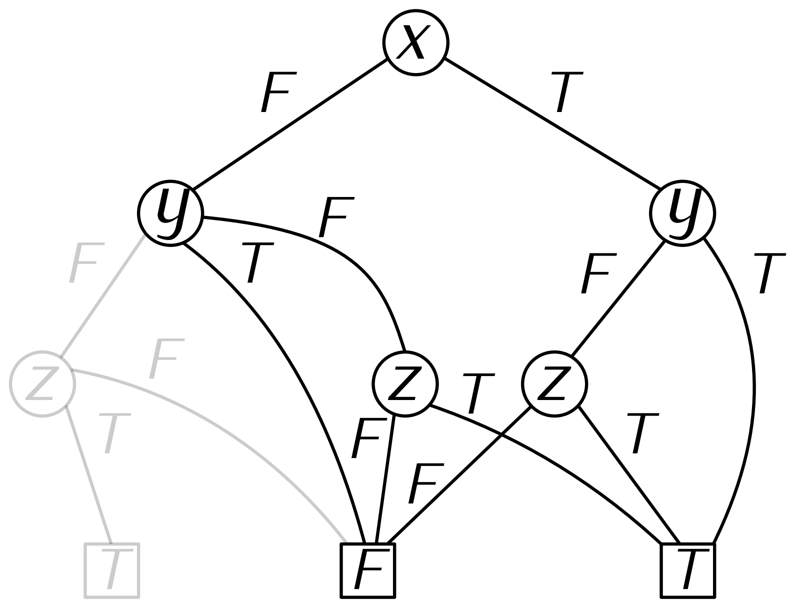
Merge more duplicate terminals

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



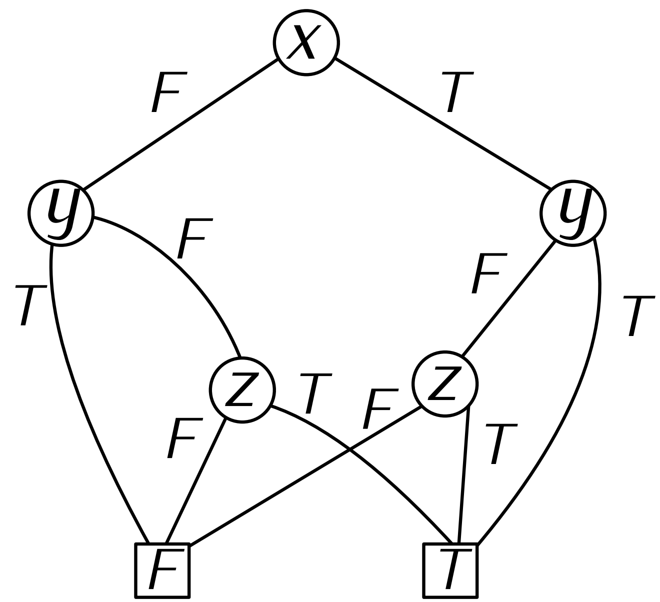
Redraw to "untangle"

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



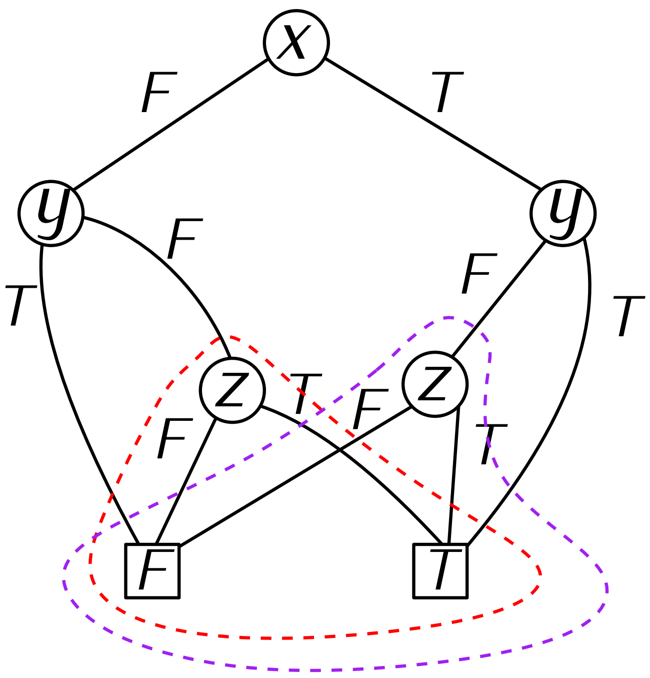
Redraw to “untangle”

Example: Reduce the given BDD for the formula $(x \land y) \lor (\neg y \land z)$ to an ROBDD.



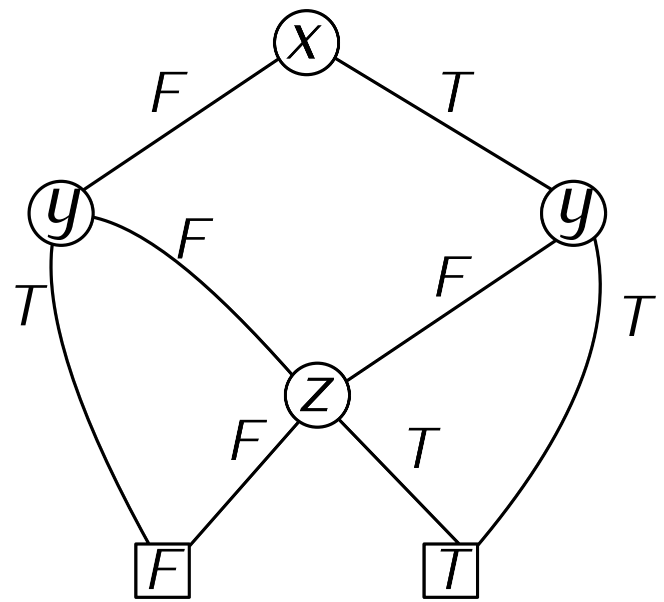
Redraw to "untangle"

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



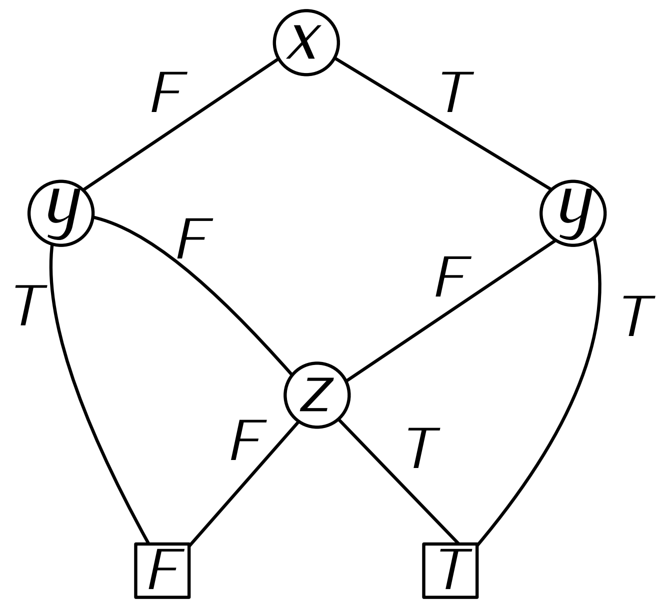
Remove duplicated sub-BDD

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



Remove duplicated sub-BDD

Example: Reduce the given BDD for the formula $(x \wedge y) \vee (\neg y \wedge z)$ to an ROBDD.



No more reduction possible

Important properties of BDDs

Ordered BDD (OBDD): variables are checked in a given order. E.g $x > y > z$.

Reduced OBDD (ROBDD): Cannot be reduced any further.

Theorem: ROBDDs are **unique** for a given ordering.

Important properties of BDDs

ROBDD **size** is sensitive to variable ordering!

Two different ROBDDs for the formula
x’ ⇔ x /\ y’ ⇔ y