

Boolean Satisfiability & Optimization

J. Marques-Silva¹

¹University College Dublin, Ireland

ECAI 2010

A Logic Puzzle I

[Russell&Norvig'03]

- If the unicorn is mythical, then it is immortal
- If the unicorn is not mythical, then it is a mortal mammal
- If the unicorn is either immortal or a mammal, then it is horned
- The unicorn is magical if it is the horned

A Logic Puzzle I

[Russell&Norvig'03]

- If the unicorn is mythical, then it is immortal
 - If the unicorn is not mythical, then it is a mortal mammal
 - If the unicorn is either immortal or a mammal, then it is horned
 - The unicorn is magical if it is the horned
-
- Is the unicorn mythical? Is it magical? Is it horned?

A Logic Puzzle II

- Propositions (can be **True (1)** or **False (0)**):
 - **M**: The unicorn is mythical
 - **I**: The unicorn is immortal
 - **L**: The unicorn is mammal
 - **H**: The unicorn is horned
 - **G**: The unicorn is magical
- If the unicorn is mythical, then it is immortal
 $M \rightarrow I$
- If the unicorn is not mythical, then it is a mortal mammal
 $\neg M \rightarrow (\neg I \wedge L)$
- If the unicorn is either immortal or a mammal, then it is horned
 $(I \vee L) \rightarrow H$
- The unicorn is magical if it is the horned
 $H \rightarrow G$

A Logic Puzzle III

- Putting it all together:
 - If the unicorn is mythical, then it is immortal
 - If the unicorn is not mythical, then it is a mortal mammal
 - If the unicorn is either immortal or a mammal, then it is horned
 - The unicorn is magical if it is horned
 - M: The unicorn is mythical
 - I: The unicorn is immortal
 - L: The unicorn is mammal
 - H: The unicorn is horned
 - G: The unicorn is magical

$$(M \rightarrow I) \wedge (\neg M \rightarrow (\neg I \wedge L)) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G)$$

A Logic Puzzle IV

- Is the unicorn **mythical**? Is it **magical**? Is it **horned**?

$$(M \rightarrow I) \wedge (\neg M \rightarrow (\neg I \wedge L)) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G) \quad \vdash$$

$$(\neg M \vee I) \wedge (M \vee (\neg I \wedge L)) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G) \quad \vdash$$

$$(\neg M \vee I) \wedge (M \vee \neg I) \wedge (M \vee L) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G) \quad \vdash$$

$$(I \vee L) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G) \quad \vdash$$

$$H \wedge G$$

- Hence, the unicorn is not necessarily mythical, but it is **horned** and **magical** !

A Logic Puzzle IV

- Is the unicorn mythical? Is it magical? Is it horned?

$$(M \rightarrow I) \wedge (\neg M \rightarrow (\neg I \wedge L)) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G) \quad \vdash$$

$$(\neg M \vee I) \wedge (M \vee (\neg I \wedge L)) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G) \quad \vdash$$

$$(\neg M \vee I) \wedge (M \vee \neg I) \wedge (M \vee L) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G) \quad \vdash$$

$$(I \vee L) \wedge ((I \vee L) \rightarrow H) \wedge (H \rightarrow G) \quad \vdash$$

$$H \wedge G$$

- Hence, the unicorn is not necessarily mythical, but it is horned and magical !
- NOTE:** Boolean formula manipulation is often impractical
 - Formulas obtained from real-world applications can easily have hundreds of thousands of variables

What is Boolean Satisfiability (SAT) ?

- Boolean formula φ is defined over a set of propositional variables $x_1, \dots, x_n$, using the standard propositional connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and parenthesis
 - The domain of propositional variables is $\{0, 1\}$
 - Example: $\varphi(x_1, \dots, x_3) = ((\neg x_1 \wedge x_2) \vee x_3) \wedge (\neg x_2 \vee x_3)$

What is Boolean Satisfiability (SAT) ?

- Boolean formula φ is defined over a set of propositional variables $x_1, \dots, x_n$, using the standard propositional connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and parenthesis
 - The domain of propositional variables is $\{0, 1\}$
 - Example: $\varphi(x_1, \dots, x_3) = ((\neg x_1 \wedge x_2) \vee x_3) \wedge (\neg x_2 \vee x_3)$
- A formula φ in **conjunctive normal form (CNF)** is a conjunction of disjunctions (**clauses**) of **literals**, where a literal is a variable or its complement
 - Example: $\varphi(x_1, \dots, x_3) = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$
- Can encode **any** Boolean formula into CNF

[Tseitin'68]

What is Boolean Satisfiability (SAT) ?

- Boolean formula φ is defined over a set of propositional variables $x_1, \dots, x_n$, using the standard propositional connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, and parenthesis
 - The domain of propositional variables is $\{0, 1\}$
 - Example: $\varphi(x_1, \dots, x_3) = ((\neg x_1 \wedge x_2) \vee x_3) \wedge (\neg x_2 \vee x_3)$
- A formula φ in **conjunctive normal form (CNF)** is a conjunction of disjunctions (**clauses**) of **literals**, where a literal is a variable or its complement
 - Example: $\varphi(x_1, \dots, x_3) = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$
- Can encode **any** Boolean formula into CNF
- The **Boolean satisfiability (SAT)** problem:
 - Find an assignment to the variables $x_1, \dots, x_n$ such that $\varphi(x_1, \dots, x_n) = 1$, or prove that no such assignment exists

[Tseitin'68]

Boolean Satisfiability (SAT)

- In theory: NP-complete

[Cook'71]

Boolean Satisfiability (SAT)

- In theory: NP-complete
- In practice: success story of Computer Science

[Cook'71]

Boolean Satisfiability (SAT)

- In theory: NP-complete [Cook'71]
- In practice: success story of Computer Science
 - Remarkable improvements since the mid 90s: Clause learning; UIPs; Search restarts; Lazy data structures; Adaptive branching heuristics; Clause minimization; Preprocessing; etc.

Boolean Satisfiability (SAT)

- In theory: NP-complete [Cook'71]
- In practice: success story of Computer Science
 - Remarkable improvements since the mid 90s: Clause learning; UIPs; Search restarts; Lazy data structures; Adaptive branching heuristics; Clause minimization; Preprocessing; etc.
 - Hundreds (thousands?) of practical applications: Hardware model checking; Software model checking; Termination analysis of term-rewrite systems; Test pattern generation (testing of software & hardware); Model finding; Symbolic trajectory evaluation; Planning; Knowledge representation; Games (n-queens, sudoku, etc.); Haplotype inference; Pedigree checking; Equivalence checking; Delay computation; Fault diagnosis; Digital filter design; Noise analysis; Cryptanalysis; Inversion attacks on hash functions; Graph coloring; Traveling salesperson; van der Waerden numbers; (your favourite SAT application here!); etc.; etc.

Boolean Satisfiability (SAT)

- In theory: NP-complete
- In practice: success story of Computer Science
 - Remarkable improvements since the mid 90s: Clause learning; UIPs; Search restarts; Lazy data structures; Adaptive branching heuristics; Clause minimization; Preprocessing; etc.
 - Hundreds (thousands?) of practical applications: Hardware model checking; Software model checking; Termination analysis of term-rewrite systems; Test pattern generation (testing of software & hardware); Model finding; Symbolic trajectory evaluation; Planning; Knowledge representation; Games (n-queens, sudoku, etc.); Haplotype inference; Pedigree checking; Equivalence checking; Delay computation; Fault diagnosis; Digital filter design; Noise analysis; Cryptanalysis; Inversion attacks on hash functions; Graph coloring; Traveling salesperson; van der Waerden numbers; (your favourite SAT application here!); etc.; etc.
 - Core engine for other solvers: Pseudo-Boolean; MaxSAT, QBF; #SAT; ASP; SMT; MVL; CSP; etc.
 - Integrated into theorem provers: HOL; Isabelle; etc.

[Cook'71]

Organization of Tutorial

- Part I:
Definitions, SAT Algorithms & Modelling Techniques
 - Basic definitions
 - Overview of SAT algorithms
 - Encoding problems to clausal form
- Part II:
Computational Problems, Extensions & Applications
 - Analyze well-known SAT-related problems: model enumeration, MUSes, prime implicants, etc.
 - Overview of extensions, covering PBO & MaxSAT
 - Representative applications

Part I

Definitions, SAT Algorithms & Modelling Techniques

Outline

Preliminaries

Algorithms

SAT-Based Modelling

Basic Definitions

- Propositional variables can be assigned value 0 or 1
 - In some contexts variables may be **unassigned**
- A clause is **satisfied** if at least one of its literals is assigned value 1
$$(x_1 \vee \neg x_2 \vee \neg x_3)$$
- A clause is **unsatisfied** if all of its literals are assigned value 0
$$(x_1 \vee \neg x_2 \vee \neg x_3)$$
- A clause is **unit** if it contains one single unassigned literal and all other literals are assigned value 0
$$(x_1 \vee \neg x_2 \vee \neg x_3)$$
- A formula is **satisfied** if **all** of its clauses are satisfied
- A formula is **unsatisfied** if **at least one** of its clauses is unsatisfied

Pure Literals

- A literal is **pure** if only occurs as a positive literal or as a negate literal in a CNF formula
 - Example:
$$\varphi = (\neg x_1 \vee x_2) \wedge (x_3 \vee \neg x_2) \wedge (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$$
 - x_1 and x_3 are pure literals
- **Pure literal rule:**
Clauses containing pure literals can be removed from the formula (i.e. just assign pure literals to the values that satisfy the clauses)
 - For the example above, the resulting formula becomes:
$$\varphi = (x_4 \vee \neg x_5) \wedge (x_5 \vee \neg x_4)$$
- A reference technique until the mid 90s; nowadays seldom used

Unit Propagation

- Unit clause rule:

[Davis&Putnam, JACM'60]

Given a unit clause, its only unassigned literal **must** be assigned value 1 for the clause to be satisfied

- Example: for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, x_3 **must** be assigned value 0

- Unit propagation

Iterated application of the unit clause rule

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

Unit Propagation

- Unit clause rule:

[Davis&Putnam, JACM'60]

Given a unit clause, its only unassigned literal **must** be assigned value 1 for the clause to be satisfied

- Example: for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, x_3 **must** be assigned value 0

- Unit propagation

Iterated application of the unit clause rule

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

Unit Propagation

- Unit clause rule:

[Davis&Putnam, JACM'60]

Given a unit clause, its only unassigned literal **must** be assigned value 1 for the clause to be satisfied

- Example: for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, x_3 **must** be assigned value 0

- Unit propagation

Iterated application of the unit clause rule

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

Unit Propagation

- Unit clause rule:

[Davis&Putnam, JACM'60]

Given a unit clause, its only unassigned literal **must** be assigned value 1 for the clause to be satisfied

- Example: for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, x_3 **must** be assigned value 0

- Unit propagation

Iterated application of the unit clause rule

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$$

Unit Propagation

- Unit clause rule:

[Davis&Putnam, JACM'60]

Given a unit clause, its only unassigned literal **must** be assigned value 1 for the clause to be satisfied

- Example: for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, x_3 **must** be assigned value 0

- Unit propagation

Iterated application of the unit clause rule

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$$

Unit Propagation

- Unit clause rule:

[Davis&Putnam, JACM'60]

Given a unit clause, its only unassigned literal **must** be assigned value 1 for the clause to be satisfied

- Example: for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, x_3 **must** be assigned value 0

- Unit propagation

Iterated application of the unit clause rule

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$$

Unit Propagation

- Unit clause rule:

[Davis&Putnam, JACM'60]

Given a unit clause, its only unassigned literal **must** be assigned value 1 for the clause to be satisfied

- Example: for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, x_3 **must** be assigned value 0

- Unit propagation

Iterated application of the unit clause rule

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4)$$

- Unit propagation can **satisfy** clauses but can also **unsatisfy** clauses (i.e. **conflicts**)

Resolution

- Resolution rule:

- If a formula φ contains clauses $(x \vee \alpha)$ and $(\neg x \vee \beta)$, then infer $(\alpha \vee \beta)$

$$\text{RES}(x \vee \alpha, \neg x \vee \beta) = (\alpha \vee \beta)$$

- Resolution forms the basis of a complete algorithm for SAT

- Iteratively apply the following steps: [Davis&Putnam, JACM'60]
 - ▶ Select variable x
 - ▶ Apply resolution rule between every pair of clauses of the form $(x \vee \alpha)$ and $(\neg x \vee \beta)$
 - ▶ Remove all clauses containing either x or $\neg x$
 - ▶ Apply the pure literal rule and unit propagation
- Terminate when either the **empty clause** or the **empty formula** is derived

Resolution – An Example

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \vdash$$

Resolution – An Example

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

Resolution – An Example

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(x_3 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

Resolution – An Example

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(x_3 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

Resolution – An Example

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(\neg x_2 \vee \neg x_3) \wedge (x_2 \vee x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(x_3 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(x_3 \vee x_4) \wedge (x_3 \vee \neg x_4) \quad \vdash$$

$$(x_3)$$

- Formula is SAT

Outline

Preliminaries

Algorithms

- Local Search

- The DPLL Algorithm

- Conflict-Driven Clause Learning (CDCL)

SAT-Based Modelling

Algorithms for SAT

- Incomplete algorithms (i.e. **cannot prove unsatisfiability**):
 - Local search / hill-climbing
 - Genetic algorithms
 - Simulated annealing
 - ...
- Complete algorithms (i.e. **can prove unsatisfiability**):
 - Proof system(s)
 - ▶ Natural deduction [e.g. Huth & Ryan'04]
 - ▶ Resolution
 - ▶ Stalmarck's method
 - ▶ Recursive learning
 - ▶ ...
 - Binary Decision Diagrams (BDDs)
 - Backtrack search / DPLL
 - ▶ Conflict-Driven Clause Learning (CDCL)
 - ...

Outline

Preliminaries

Algorithms

- Local Search

- The DPLL Algorithm

- Conflict-Driven Clause Learning (CDCL)

SAT-Based Modelling

Organization of Local Search

- Local search is incomplete; it **cannot** prove unsatisfiability
 - Very effective in specific contexts
- Example:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

Organization of Local Search

- Local search is incomplete; it **cannot** prove unsatisfiability
 - Very effective in specific contexts
- Example:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

- Start with (possibly random) assignment: $x_4 = 0, x_1 = x_2 = x_3 = 1$
- And repeat a number of times:

Organization of Local Search

- Local search is incomplete; it **cannot** prove unsatisfiability
 - Very effective in specific contexts
- Example:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

- Start with (possibly random) assignment: $x_4 = 0, x_1 = x_2 = x_3 = 1$
- And repeat a number of times:

Organization of Local Search

- Local search is incomplete; it **cannot** prove unsatisfiability
 - Very effective in specific contexts
- Example:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

- Start with (possibly random) assignment: $x_4 = 0, x_1 = x_2 = x_3 = 1$
- And repeat a number of times:
 - If not all clauses satisfied, flip variable (e.g. x_4)

Organization of Local Search

- Local search is incomplete; it **cannot** prove unsatisfiability
 - Very effective in specific contexts
- Example:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

- Start with (possibly random) assignment: $x_4 = 0, x_1 = x_2 = x_3 = 1$
- And repeat a number of times:
 - If not all clauses satisfied, flip variable (e.g. x_4)

Organization of Local Search

- Local search is incomplete; it **cannot** prove unsatisfiability
 - Very effective in specific contexts
- Example:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

- Start with (possibly random) assignment: $x_4 = 0, x_1 = x_2 = x_3 = 1$
- And repeat a number of times:
 - If not all clauses satisfied, flip variable (e.g. x_4)
 - Done if all clauses satisfied

Organization of Local Search

- Local search is incomplete; it **cannot** prove unsatisfiability
 - Very effective in specific contexts
- Example:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

- Start with (possibly random) assignment: $x_4 = 0, x_1 = x_2 = x_3 = 1$
- And repeat a number of times:
 - If not all clauses satisfied, flip variable (e.g. x_4)
 - Done if all clauses satisfied
- Repeat (random) selection of assignment a number of times

Outline

Preliminaries

Algorithms

Local Search

The DPLL Algorithm

Conflict-Driven Clause Learning (CDCL)

SAT-Based Modelling

DPLL – Historical Perspective

- In 1960, M. Davis and H. Putnam proposed the DP algorithm:
 - Resolution used to eliminate 1 variable at each step
 - Applied the pure literal rule and unit propagation
- Original algorithm was inefficient
- In 1962, M. Davis, G. Logemann and D. Loveland proposed an alternative algorithm:
 - Instead of eliminating variables, the algorithm would split on a given variable at each step
 - Also applied the pure literal rule and unit propagation
- The 1962 algorithm is actually an implementation of **backtrack search**
- Over the years the 1962 algorithm became known as the DPLL (sometimes DLL) algorithm

The DPLL Algorithm

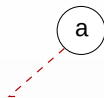
- Standard **backtrack search**
- At each step:
 - **[DECIDE]** Select decision assignment
 - **[DEDUCE]** Apply unit propagation and (optionally) the pure literal rule
 - **[DIAGNOSE]** If conflict identified, then backtrack
 - ▶ If cannot backtrack further, return **UNSAT**
 - ▶ Otherwise, proceed with unit propagation
 - If formula satisfied, return **SAT**
 - Otherwise, proceed with another decision

An Example of DPLL

$$\begin{aligned}\varphi = & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$

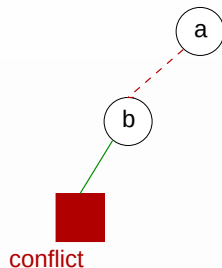
An Example of DPLL

$$\begin{aligned}\varphi = & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



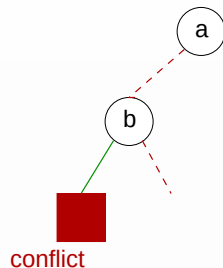
An Example of DPLL

$$\begin{aligned}\varphi = & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



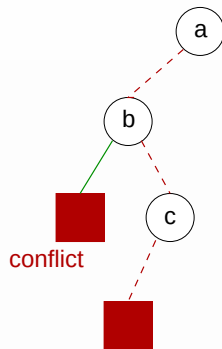
An Example of DPLL

$$\begin{aligned}\varphi = & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



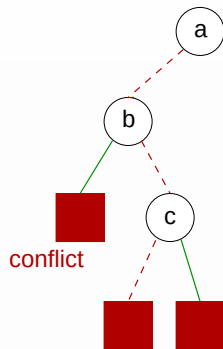
An Example of DPLL

$$\begin{aligned}\varphi = & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



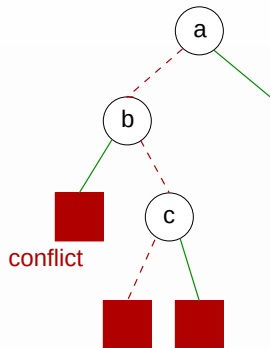
An Example of DPLL

$$\begin{aligned}\varphi = & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



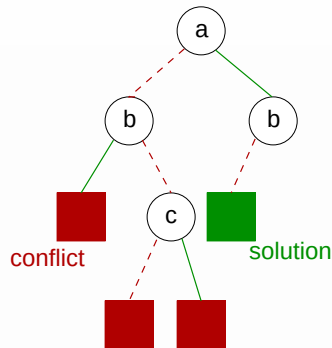
An Example of DPLL

$$\begin{aligned}\varphi = & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



An Example of DPLL

$$\begin{aligned}\varphi = & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



Outline

Preliminaries

Algorithms

Local Search

The DPLL Algorithm

Conflict-Driven Clause Learning (CDCL)

SAT-Based Modelling

CDCL SAT Solvers – Basic Techniques

- Based on DPLL [Davis et al., JACM'60, CACM'62]
 - Must be able to prove unsatisfiability
- New clauses are **learned** from conflicts [Marques-Silva&Sakallah, ICCAD'96]
 - Backtracking can be **non-chronological**
- Structure of conflicts is exploited (**UIPs**) [Marques-Silva&Sakallah, ICCAD'96]
- Backtrack search is periodically **restarted** [Gomes et al., AAAI'98]
- Lazy data structures are used [Moskewicz et al, DAC'01]
 - Compact with low maintenance overhead
- Branching is guided by conflicts [Moskewicz et al, DAC'01]
 - E.g. VSIDS, etc.

CDCL SAT Solvers – Additional Techniques

- (Currently) **effective** techniques:

- Unused learned clauses are discarded
- Use formula preprocessing I
- Minimize learned clauses
- Use literal progress saving
- Use dynamic restart policies
- Exploit **extended implication graphs**
- Identify **glue** clauses

[Goldberg&Novikov, DATE'02]

[Een&Biere, SAT'05]

[Sorensson&Biere, SAT'09]

[Pipatsrisawat&Darwiche, SAT'07]

[Biere, SAT'08]

[Audemard et al., SAT'08]

[Audemard & Simon, IJCAI'09]

- (Currently) **ineffective** techniques:

- Identify pure literals
- Implement variable lookahead
- Use formula preprocessing II

[Davis&Putnam, JACM'60]

[Anbulagan&Li, IJCAI'97]

[Brafman, IJCAI'01]

Clause Learning

- During backtrack search, for each conflict **learn new clause**, which **explains** and **prevents** repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

Clause Learning

- During backtrack search, for each conflict **learn new clause**, which **explains** and **prevents** repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decisions $c = 0$ and $f = 0$

Clause Learning

- During backtrack search, for each conflict **learn new clause**, which **explains** and **prevents** repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decisions $c = 0$ and $f = 0$
- Assign $a = 0$ and imply assignments

Clause Learning

- During backtrack search, for each conflict **learn new clause**, which **explains** and **prevents** repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decisions $c = 0$ and $f = 0$
- Assign $a = 0$ and imply assignments

Clause Learning

- During backtrack search, for each conflict **learn new clause**, which **explains** and **prevents** repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decisions $c = 0$ and $f = 0$
- Assign $a = 0$ and imply assignments
- A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied

Clause Learning

- During backtrack search, for each conflict **learn new clause**, which **explains** and **prevents** repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decisions $c = 0$ and $f = 0$
- Assign $a = 0$ and imply assignments
- A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
- $(a = 0) \wedge (c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$

Clause Learning

- During backtrack search, for each conflict **learn new clause**, which **explains** and **prevents** repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decisions $c = 0$ and $f = 0$
- Assign $a = 0$ and imply assignments
- A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
- $(a = 0) \wedge (c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$
- $(\varphi = 1) \Rightarrow (a = 1) \vee (c = 1) \vee (f = 1)$

Clause Learning

- During backtrack search, for each conflict **learn new clause**, which **explains** and **prevents** repetition of the same conflict

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \dots$$

- Assume decisions $c = 0$ and $f = 0$
- Assign $a = 0$ and imply assignments
- A conflict is reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
- $(a = 0) \wedge (c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$
- $(\varphi = 1) \Rightarrow (a = 1) \vee (c = 1) \vee (f = 1)$
- Learn new clause $(a \vee c \vee f)$

Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

$$\begin{aligned}\varphi = & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

$$\begin{aligned}\varphi = & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

- Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$

Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
- Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$

Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
- Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$

Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
- Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$

Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
- Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$
- A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied

Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

$$\begin{aligned}\varphi = & (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ & (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)\end{aligned}$$

- Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
- Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$
- A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
- $(c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$

Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
- Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$
- A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
- $(c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$
- $(\varphi = 1) \Rightarrow (c = 1) \vee (f = 1)$

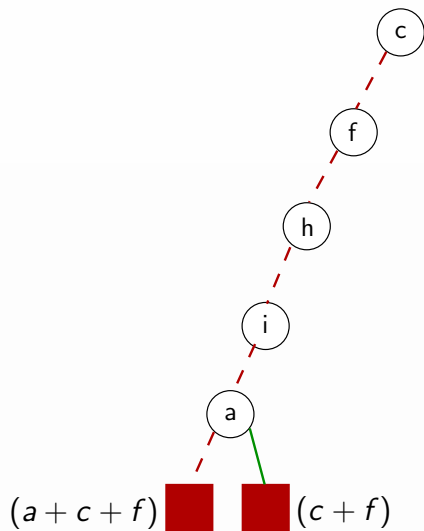
Non-Chronological Backtracking

- During backtrack search, for each conflict **backtrack to one of the causes of the conflict**

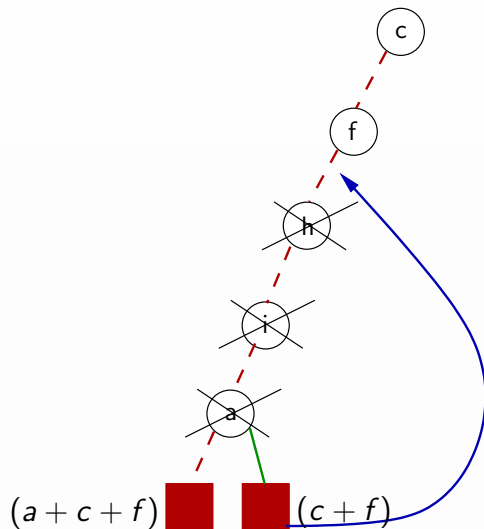
$$\varphi = (a \vee b) \wedge (\neg b \vee c \vee d) \wedge (\neg b \vee e) \wedge (\neg d \vee \neg e \vee f) \wedge \\ (a \vee c \vee f) \wedge (\neg a \vee g) \wedge (\neg g \vee b) \wedge (\neg h \vee j) \wedge (\neg i \vee k)$$

- Assume decisions $c = 0$, $f = 0$, $h = 0$ and $i = 0$
- Assignment $a = 0$ caused conflict $\Rightarrow$ learnt clause $(a \vee c \vee f)$ implies $a = 1$
- A conflict is again reached: $(\neg d \vee \neg e \vee f)$ is unsatisfied
- $(c = 0) \wedge (f = 0) \Rightarrow (\varphi = 0)$
- $(\varphi = 1) \Rightarrow (c = 1) \vee (f = 1)$
- Learn new clause $(c \vee f)$

Non-Chronological Backtracking

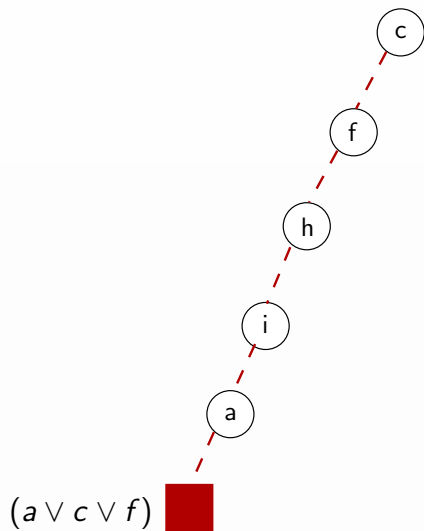


Non-Chronological Backtracking

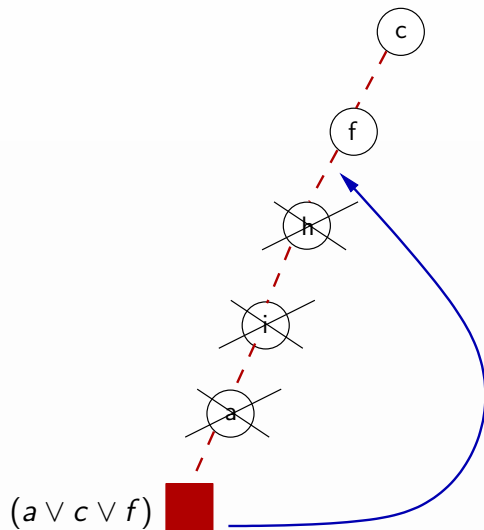


- Learnt clause: $(c \vee f)$
- Need to backtrack, given new clause
- Backtrack to most recent decision: $f = 0$
- Clause learning and non-chronological backtracking are hallmarks of modern SAT solvers

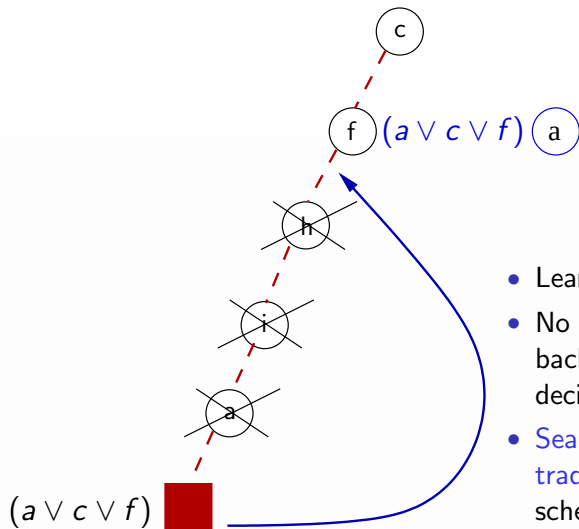
Most Recent Backtracking Scheme



Most Recent Backtracking Scheme

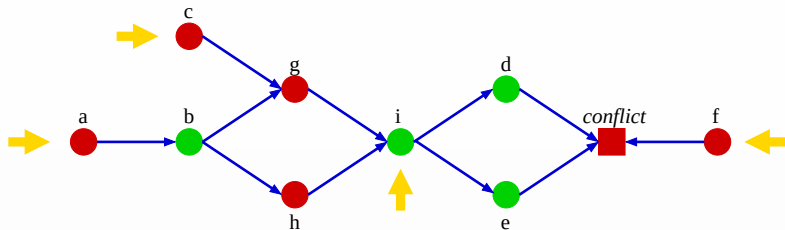


Most Recent Backtracking Scheme



- Learnt clause: $(a \vee c \vee f)$
- No need to assign $a = 1$:
backtrack to most recent
decision: $f = 0$
- Search algorithm is no longer a
traditional backtracking
scheme
- Akin to dynamic backtracking

Unique Implication Points (UIPs)



- Exploit **structure** from the implication graph
 - To have a more aggressive backtracking policy
- Identify **additional clauses** to learn
 - Create clauses $(a \vee c \vee f)$ and $(\neg i \vee f)$
 - Imply not only $a = 1$ but also $i = 0$
- 1st UIP scheme is the most effective
 - Create only one clause $(\neg i \vee f)$
 - Avoid creating similar clauses involving the same literals

[Marques-Silva&Sakallah'96]

[Zhanget al.'01]

Clause deletion policies

- Keep only the **small clauses**

[Marques-Silva&Sakallah'96]

- For each conflict record one clause
- Keep clauses of size $\leq K$
- Large clauses get deleted when become unresolved

- Keep only the **relevant clauses**

[Bayardo&Schrag'97]

- Delete unresolved clauses with $\leq M$ free literals

- Keep only the clauses **that are used**

[Goldberg&Novikov'02]

- Keep track of clauses **activity**

Data Structures

- **Key point:** only unit and unsatisfied clauses *must* be detected during search
 - Formula is **unsatisfied** when at least one clause is unsatisfied
 - Formula is **satisfied** when all the variables are assigned and there are no unsatisfied clauses
- **In practice:** unit and unsatisfied clauses may be identified using only **two references**
- Standard data structures (**adjacency lists**):
 - Each variable x keeps a reference to **all** clauses containing a literal in x
- Lazy data structures (**watched literals**):
 - For each clause, only two variables keep a reference to the clause, i.e. only 2 literals are **watched**

Standard Data Structures (adjacency lists)

literals0 = 4

literals1 = 0

size = 5



unit

literals0 = 4

literals1 = 1

size = 5



satisfied

literals0 = 5

literals1 = 0

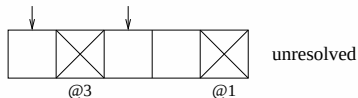
size = 5



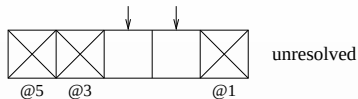
unsatisfied

- Each variable x keeps a reference to **all** clauses containing a literal in x
 - If variable x is assigned, then **all** clauses containing a literal in x are evaluated
 - If search backtracks, then **all** clauses of all newly unassigned variables are updated
- Total number of references is L , where L is the number of literals

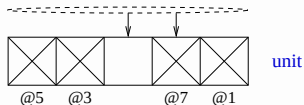
Lazy Data Structures (watched literals)



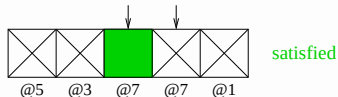
unresolved



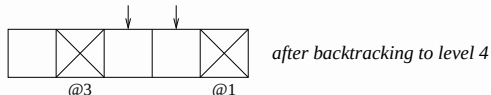
unresolved



unit



satisfied



after backtracking to level 4

- For each clause, only two variables keep a reference to the clause, i.e. only 2 literals are **watched**
 - If variable x is assigned, **only** the clauses where literals in x are watched need to be evaluated
 - If search backtracks, then **nothing** needs to be done
- Total number of references is $2 \times C$, where C is the number of clauses
 - In general $L \gg 2 \times C$, in particular if clauses are learnt

Search Heuristics

- Standard data structures: heavy heuristics

- DLIS: Dynamic Large Individual Sum

[Marques-Silva'99]

- ▶ Selects the literal that appears most frequently in unresolved clauses

- Lazy data structures: light heuristics

- VSIDS: Variable State Independent Decaying Sum

[Moskewicz et al.'01]

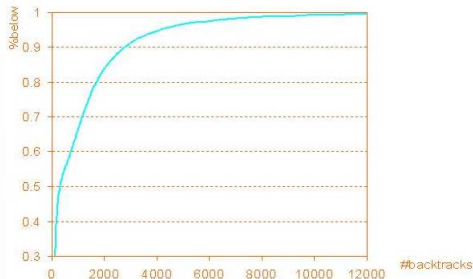
- ▶ Each literal has a counter, initialized to zero
 - ▶ When a new clause is recorded, the counter associated with each literal in the clause is incremented
 - ▶ The unassigned literal with the highest counter is chosen at each decision

- Examples of variants

- ▶ Counters updated also for literals in the clauses involved in conflicts

[Goldberg&Novikov'02]

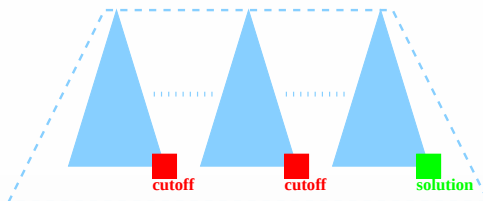
Restarts I



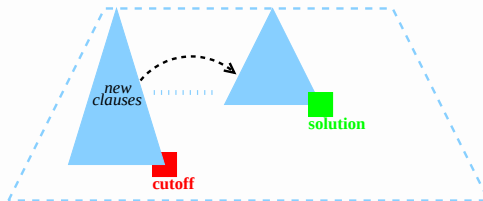
- Plot for **processor verification instance** with **branching randomization** and 10000 runs
 - More than 50% of the runs require less than 1000 backtracks
 - A small percentage requires more than 10000 backtracks
- Run times of backtrack search SAT solvers characterized by **heavy-tail distributions**

[Gomes et al.'98]

Restarts II



- Repeatedly restart the search each time a **cutoff** is reached
 - Randomization allows to explore different paths in search tree
- Resulting algorithm is incomplete
 - Increase the cutoff value
 - Keep clauses from previous runs



Evolution of SAT Solvers

Instance	Posit'94	Grasp'96	Chaff'03	Minisat'03	Picosat'08
ssa2670-136	13.57	0.22	0.02	0.00	0.01
bf1355-638	310.93	0.02	0.02	0.00	0.03
design_3	> 1800	3.93	0.18	0.17	0.93
design_1	> 1800	34.55	0.35	0.11	0.68
4pipe_4_ooo	> 1800	> 1800	17.47	110.97	44.95
fifo8_300	> 1800	> 1800	348.50	53.66	39.31
w08_15	> 1800	> 1800	> 1800	99.10	71.89
9pipe_9_ooo	> 1800	> 1800	> 1800	> 1800	> 1800
c6288	> 1800	> 1800	> 1800	> 1800	> 1800

- Modern SAT algorithms can solve instances with hundreds of thousands of variables and tens of millions of clauses

Outline

Preliminaries

Algorithms

SAT-Based Modelling

- Boolean Formulas

- Cardinality Constraints

- Pseudo-Boolean Constraints

- Encoding CSPs

Outline

Preliminaries

Algorithms

SAT-Based Modelling

Boolean Formulas

Cardinality Constraints

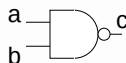
Pseudo-Boolean Constraints

Encoding CSPs

Representing Boolean Formulas / Circuits I

- Satisfiability problems can be defined on Boolean circuits/formulas
- Can represent circuits/formulas as CNF formulas [Tseitin'68]
 - For each (simple) gate, CNF formula encodes the **consistent** assignments to the gate's inputs and output
 - ▶ Given $z = OP(x, y)$, represent in CNF $z \leftrightarrow OP(x, y)$
 - CNF formula for the circuit is the conjunction of CNF formula for each gate

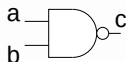
$$\varphi_c = (a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$



$$\varphi_t = (\neg r \vee t) \wedge (\neg s \vee t) \wedge (r \vee s \vee \neg t)$$



Representing Boolean Formulas / Circuits II

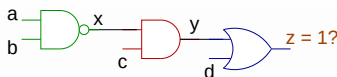


a	b	c	$\varphi_c(a,b,c)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$$\varphi_c = (a \vee c) \wedge (b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

Representing Boolean Formulas / Circuits III

- CNF formula for the circuit is the conjunction of the CNF formula for each gate
 - Can specify objectives with additional clauses



$$\begin{aligned}\varphi = & (a \vee x) \wedge (b \vee x) \wedge (\neg a \vee \neg b \vee \neg x) \wedge \\ & (x \vee \neg y) \wedge (c \vee \neg y) \wedge (\neg x \vee \neg c \vee y) \wedge \\ & (\neg y \vee z) \wedge (\neg d \vee z) \wedge (y \vee d \vee \neg z) \wedge \\ & (z)\end{aligned}$$

- Note: $z = d \vee (c \wedge (\neg(a \wedge b)))$
 - No distinction between Boolean circuits and formulas

Representing Boolean Formulas / Circuits IV

- Example CNF encodings:

$$z = \text{AND}(x, y) \quad (\neg x \vee \neg y \vee z) \wedge (x \vee \neg z) \wedge (y \vee \neg z)$$

$$z = \text{NAND}(x, y) \quad (\neg x \vee \neg y \vee \neg z) \wedge (x \vee z) \wedge (y \vee z)$$

$$z = \text{OR}(x, y) \quad (x \vee y \vee \neg z) \wedge (\neg x \vee z) \wedge (\neg y \vee z)$$

$$z = \text{NOR}(x, y) \quad (x \vee y \vee z) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg z)$$

$$z = \text{NOT}(x) \quad (x \vee z) \wedge (\neg x \vee \neg z)$$

Representing Boolean Formulas / Circuits IV

- Example CNF encodings:

$$z = \text{AND}(x, y) \quad (\neg x \vee \neg y \vee z) \wedge (x \vee \neg z) \wedge (y \vee \neg z)$$

$$z = \text{NAND}(x, y) \quad (\neg x \vee \neg y \vee \neg z) \wedge (x \vee z) \wedge (y \vee z)$$

$$z = \text{OR}(x, y) \quad (x \vee y \vee \neg z) \wedge (\neg x \vee z) \wedge (\neg y \vee z)$$

$$z = \text{NOR}(x, y) \quad (x \vee y \vee z) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee \neg z)$$

$$z = \text{NOT}(x) \quad (x \vee z) \wedge (\neg x \vee \neg z)$$

- Building blocks for more sophisticated CNF encodings

Representing Boolean Formulas / Circuits V

- Encoding for the **ITE** (if-then-else) operator $z = \text{ITE}(s, x, y)$

- Use definition:

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

- Can add redundant clauses:

[Een&Sorensson'06]

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

Representing Boolean Formulas / Circuits V

- Encoding for the **ITE** (if-then-else) operator $z = \text{ITE}(s, x, y)$

- Use definition:

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

- Can add redundant clauses:

[Een&Sorensson'06]

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

- Building block for more sophisticated CNF encodings

Outline

Preliminaries

Algorithms

SAT-Based Modelling

Boolean Formulas

Cardinality Constraints

Pseudo-Boolean Constraints

Encoding CSPs

Cardinality Constraints

- How to handle cardinality constraints, $\sum_{j=1}^n x_j \leq k$?
 - How to handle AtMost1 constraints, $\sum_{j=1}^n x_j \leq 1$?
- Solution #1:
 - Use PB solver
 - Difficult to keep up with advances in SAT technology
 - For SAT/UNSAT, best PB solvers already encode to SAT

Cardinality Constraints

- How to handle cardinality constraints, $\sum_{j=1}^n x_j \leq k$?
 - How to handle AtMost1 constraints, $\sum_{j=1}^n x_j \leq 1$?
- Solution #1:
 - Use PB solver
 - Difficult to keep up with advances in SAT technology
 - For SAT/UNSAT, best PB solvers already encode to SAT
- Solution #2:
 - Encode cardinality constraints to CNF
 - Use SAT solver

Equals1, AtLeast1 & AtMost1 Constraints

- $\sum_{j=1}^n x_j = 1$: encode with $(\sum_{j=1}^n x_j \leq 1) \wedge (\sum_{j=1}^n x_j \geq 1)$
- $\sum_{j=1}^n x_j \geq 1$: encode with $(x_1 \vee x_2 \vee \dots \vee x_n)$
- $\sum_{j=1}^n x_j \leq 1$ encode with:
 - Pairwise encoding
 - ▶ Clauses: $\mathcal{O}(n^2)$; No auxiliary variables
 - Sequential counter [Sinz'05]
 - ▶ Clauses: $\mathcal{O}(n)$; Auxiliary variables: $\mathcal{O}(n)$
 - Bitwise encoding [Prestwich'07]
 - ▶ Clauses: $\mathcal{O}(n \log n)$; Auxiliary variables: $\mathcal{O}(\log n)$
 - ...

Sequential Counter Encoding

- Encode $\sum_{j=1}^n x_j \leq 1$ with **sequential counter**:

$$\begin{aligned} & (\neg x_1 \vee s_1) \wedge (\neg x_n \vee \neg s_{n-1}) \wedge \\ & \bigwedge_{1 < i < n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1})) \end{aligned}$$

- If $x_j = 1$, all s_i variables assigned
 - ▶ All other x variables **must** take value 0
- If all $x_j = 0$, can find **consistent** assignment to s_i variables
- $\mathcal{O}(n)$ clauses ; $\mathcal{O}(n)$ auxiliary variables

Bitwise Encoding

- Encode $\sum_{j=1}^n x_j \leq 1$ with bitwise encoding:
 - Define $r = \lceil \log n \rceil$ (with $n > 1$) ; Auxiliary variables $v_0, \dots, v_{r-1}$
 - Associate with x_j the binary representation of $j - 1$
 - Create clauses $(\neg x_j \vee p_i)$, $i = 0, \dots, r - 1$, where
 - ▶ $p_i = v_i$ if the binary representation of $j - 1$ has value 1 in position i
 - ▶ $p_i = \neg v_i$ otherwise
 - If $x_j = 1$, assignment to v_i variables **must** encode $j - 1$
 - ▶ All other x variables take value 0
 - If all $x_j = 0$, **any** assignment to v_i variables is consistent
 - $\mathcal{O}(n \log n)$ clauses ; $\mathcal{O}(\log n)$ auxiliary variables
- An example: $x_1 + x_2 + x_3 \leq 1$

	$j - 1$	$v_1 v_0$
x_1	0	00
x_2	1	01
x_3	2	10

$$\begin{aligned} &(\neg x_1 \vee \neg v_1) \wedge (\neg x_1 \vee \neg v_0) \\ &(\neg x_2 \vee \neg v_1) \wedge (\neg x_2 \vee v_0) \\ &(\neg x_3 \vee v_1) \wedge (\neg x_3 \vee \neg v_0) \end{aligned}$$

General Cardinality Constraints

- General form: $\sum_{j=1}^n x_j \leq k$
 - Sequential counter
 - ▶ Clauses/Variables: $\mathcal{O}(n k)$
 - BDDs
 - ▶ Clauses/Variables: $\mathcal{O}(n k)$
 - Sorting networks
 - ▶ Clauses/Variables: $\mathcal{O}(n \log^2 n)$
 - Cardinality Networks:
 - ▶ Clauses/Variables: $\mathcal{O}(n \log^2 k)$
 - ...

[Sinz'05]

[Een&Sorensson'06]

[Een&Sorensson'06]

[Asin et al.'09]

Sorting Networks I

- Encode $\sum_{j=1}^n x_j \leq k$ with sorting network:

- Unary representation
- Use **odd-even merging networks**
- Recursive definition

[Batcher'68, Een&Sorensson'06, Asin et al.'09]

Sorting Networks I

- Encode $\sum_{j=1}^n x_j \leq k$ with sorting network:

- Unary representation
- Use **odd-even merging networks**
- Recursive definition

[Batcher'68, Een&Sorensson'06, Asin et al.'09]

- ▶ Base Case:

$$\text{Merge}(a_1, b_1) \triangleq (\langle c_1, c_2 \rangle, \{c_2 = \min(a_1, b_1), c_1 = \max(a_1, b_1)\})$$

Sorting Networks I

- Encode $\sum_{j=1}^n x_j \leq k$ with sorting network:

- Unary representation
- Use **odd-even merging networks**
- Recursive definition

[Batcher'68, Eén&Sörensson'06, Asin et al.'09]

- ▶ Base Case:

$$\text{Merge}(a_1, b_1) \triangleq (\langle c_1, c_2 \rangle, \{c_2 = \min(a_1, b_1), c_1 = \max(a_1, b_1)\})$$

- ▶ Let:

$$\text{Merge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle) \triangleq (\langle d_1, \dots, d_n \rangle, S_{\text{odd}})$$

$$\text{Merge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle) \triangleq (\langle e_1, \dots, e_n \rangle, S_{\text{even}})$$

Sorting Networks I

- Encode $\sum_{j=1}^n x_j \leq k$ with sorting network:

- Unary representation
- Use **odd-even merging networks**
- Recursive definition

[Batcher'68, Een&Sorensson'06, Asin et al.'09]

- ▶ Base Case:

$$\text{Merge}(a_1, b_1) \triangleq (\langle c_1, c_2 \rangle, \{c_2 = \min(a_1, b_1), c_1 = \max(a_1, b_1)\})$$

- ▶ Let:

$$\text{Merge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle) \triangleq (\langle d_1, \dots, d_n \rangle, S_{\text{odd}})$$

$$\text{Merge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle) \triangleq (\langle e_1, \dots, e_n \rangle, S_{\text{even}})$$

- ▶ Then:

$$\text{Merge}(\langle a_1, a_2, \dots, a_n \rangle, \langle b_1, b_2, \dots, b_n \rangle) \triangleq$$

$$(\langle d_1, c_1, \dots, c_{2n-1}, e_n \rangle, S_{\text{odd}} \cup S_{\text{even}} \cup S_{\text{mrg}})$$

- ▶ Where:

$$S_{\text{mrg}} = \bigcup_{i=1}^{n-1} \{c_{2i+1} = \min(d_{i+1}, e_i), c_{2i} = \max(d_{i+1}, e_i)\}$$

Sorting Networks I

- Encode $\sum_{j=1}^n x_j \leq k$ with sorting network:

- Unary representation
- Use **odd-even merging networks**
- Recursive definition

[Batcher'68, Een&Sorensson'06, Asin et al.'09]

- ▶ Base Case:

$$\text{Merge}(a_1, b_1) \triangleq (\langle c_1, c_2 \rangle, \{c_2 = \min(a_1, b_1), c_1 = \max(a_1, b_1)\})$$

- ▶ Let:

$$\text{Merge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle) \triangleq (\langle d_1, \dots, d_n \rangle, S_{\text{odd}})$$

$$\text{Merge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle) \triangleq (\langle e_1, \dots, e_n \rangle, S_{\text{even}})$$

- ▶ Then:

$$\begin{aligned} \text{Merge}(\langle a_1, a_2, \dots, a_n \rangle, \langle b_1, b_2, \dots, b_n \rangle) &\triangleq \\ (\langle d_1, c_1, \dots, c_{2n-1}, e_n \rangle, S_{\text{odd}} \cup S_{\text{even}} \cup S_{\text{mrg}}) \end{aligned}$$

- ▶ Where:

$$S_{\text{mrg}} = \bigcup_{i=1}^{n-1} \{c_{2i+1} = \min(d_{i+1}, e_i), c_{2i} = \max(d_{i+1}, e_i)\}$$

- ▶ Let $\langle z_1, \dots, z_n \rangle$ be the sorted output. The constraint is:
 $z_i = 0, \quad i > k$

Sorting Networks I

- Encode $\sum_{j=1}^n x_j \leq k$ with sorting network:

- Unary representation
- Use **odd-even merging networks**
- Recursive definition

[Batcher'68, Eén&Sorensson'06, Asin et al.'09]

- ▶ Base Case:

$$\text{Merge}(a_1, b_1) \triangleq (\langle c_1, c_2 \rangle, \{c_2 = \min(a_1, b_1), c_1 = \max(a_1, b_1)\})$$

- ▶ Let:

$$\text{Merge}(\langle a_1, a_3, \dots, a_{n-1} \rangle, \langle b_1, b_3, \dots, b_{n-1} \rangle) \triangleq (\langle d_1, \dots, d_n \rangle, S_{\text{odd}})$$

$$\text{Merge}(\langle a_2, a_4, \dots, a_n \rangle, \langle b_2, b_4, \dots, b_n \rangle) \triangleq (\langle e_1, \dots, e_n \rangle, S_{\text{even}})$$

- ▶ Then:

$$\begin{aligned} \text{Merge}(\langle a_1, a_2, \dots, a_n \rangle, \langle b_1, b_2, \dots, b_n \rangle) &\triangleq \\ (\langle d_1, c_1, \dots, c_{2n-1}, e_n \rangle, S_{\text{odd}} \cup S_{\text{even}} \cup S_{\text{mrg}}) \end{aligned}$$

- ▶ Where:

$$S_{\text{mrg}} = \bigcup_{i=1}^{n-1} \{c_{2i+1} = \min(d_{i+1}, e_i), c_{2i} = \max(d_{i+1}, e_i)\}$$

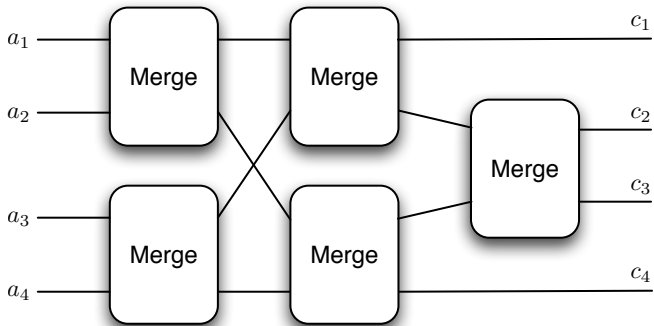
- ▶ Let $\langle z_1, \dots, z_n \rangle$ be the sorted output. The constraint is:

$$z_i = 0, \quad i > k$$

- Note: **min** $\equiv$ AND and **max** $\equiv$ OR

Sorting Networks II

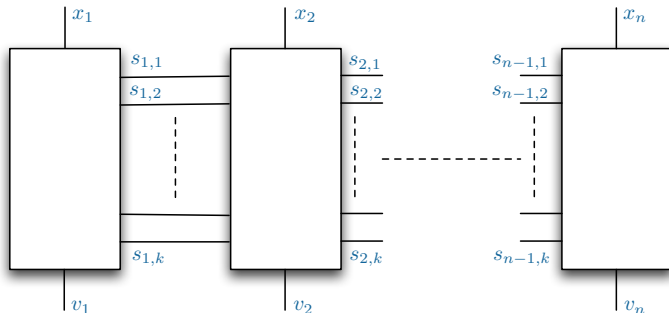
- Sort $\langle a_1, a_2, a_3, a_4 \rangle$:



where each **Merge** block contains 1 min (AND) and 1 max (OR) operators

Sequential Counter – Revisited I

- Encode $\sum_{j=1}^n x_j \leq k$ with sequential counter:



- Equations for each block $1 < i < n$, $1 < j < k$:

$$s_i = \sum_{j=1}^i x_j$$

s_i represented in unary

$$s_{i,1} = s_{i-1,1} \vee x_i$$

$$s_{i,j} = s_{i-1,j} \vee s_{i-1,j-1} \wedge x_i$$

$$v_i = s_{i-1,k} \wedge x_i = 0$$

Sequential Counter – Revisited II

- CNF formula for $\sum_{j=1}^n x_j \leq k$:
 - Assume: $k > 0 \wedge n > 1$
 - Indexes: $1 < i < n$, $1 < j \leq k$

$$\begin{aligned} &(\neg x_1 \vee x_{1,1}) \\ &(\neg s_{1,j}) \\ &(\neg x_i \vee s_{i,1}) \\ &(\neg s_{i-1,1} \vee s_{i,1}) \\ &(\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j}) \\ &(\neg s_{i-1,j} \vee s_{i,j}) \\ &(\neg x_i \vee \neg s_{i-1,k}) \\ &(\neg x_n \vee \neg s_{n-1,k}) \end{aligned}$$

- Recall: $\mathcal{O}(n k)$ clauses & variables

Outline

Preliminaries

Algorithms

SAT-Based Modelling

Boolean Formulas

Cardinality Constraints

Pseudo-Boolean Constraints

Encoding CSPs

Pseudo-Boolean Constraints

- General form: $\sum_{j=1}^n a_j \cdot x_j \leq b$

- Operational encoding

[Warners'98]

- ▶ Clauses/Variables: $\mathcal{O}(n)$
- ▶ Does **not** guarantee arc-consistency

- BDDs

[Een&Sorensson'06]

- ▶ Worst-case exponential number of clauses

- Polynomial watchdog encoding

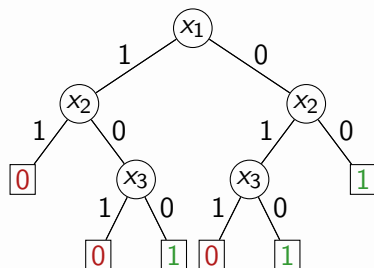
[Bailleux, Boufkhad & Roussel'09]

- ▶ Let $\nu(n) = \log(n) \log(a_{\max})$
- ▶ Clauses: $\mathcal{O}(n^3 \nu(n))$; Aux variables: $\mathcal{O}(n^2 \nu(n))$

- ...

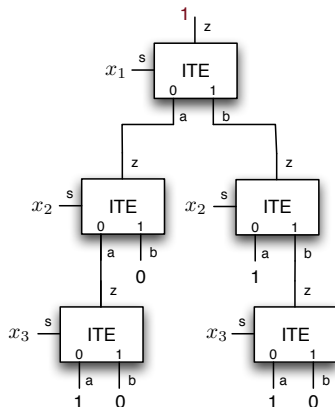
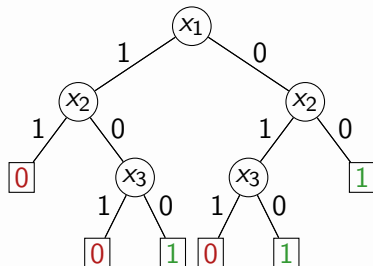
Encoding PB Constraints with BDDs I

- Encode $3x_1 + 3x_2 + x_3 \leq 3$
- Construct BDD
 - E.g. by analyzing variables by decreasing coefficients
- Extract ITE-based circuit from BDD



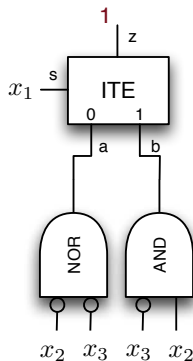
Encoding PB Constraints with BDDs I

- Encode $3x_1 + 3x_2 + x_3 \leq 3$
- Construct BDD
 - E.g. by analyzing variables by decreasing coefficients
- Extract ITE-based circuit from BDD



Encoding PB Constraints with BDDs II

- Encode $3x_1 + 3x_2 + x_3 \leq 3$
- Extract ITE-based circuit from BDD
- Simplify and create final circuit:



Outline

Preliminaries

Algorithms

SAT-Based Modelling

Boolean Formulas

Cardinality Constraints

Pseudo-Boolean Constraints

Encoding CSPs

CSP Constraints

- Many possible encodings:

- Direct encoding

[de Kleer'89, Genisson&Jegou'96, Walsh'00]

- Log encoding

[Walsh'00]

- Support encoding

[Kasif'90, Gent'02]

- Log-Support encoding

[Gavanelli'07]

Direct Encoding for CSP w/ Binary Constraints

- Variable x_i with domain D_i , with $m_i = |D_i|$
- Represent values of x_i with Boolean variables $x_{i,1}, \dots, x_{i,m_i}$
- Require $\sum_{k=1}^{m_i} x_{i,k} = 1$
 - Suffices to require $\sum_{k=1}^{m_i} x_{i,k} \geq 1$
- If the pair of assignments $x_i = v_i \wedge x_j = v_j$ is not allowed, add binary clause $(\neg x_{i,v_i} \vee \neg x_{j,v_j})$

[Walsh'00]

Part II

Computational Problems, Extensions & Applications

Outline

SAT-Related Computational Problems

- Model Enumeration

- Minimally Unsatisfiable Subformulas

- Prime Implicant & Minimal Models

Extensions of SAT

Example Applications

Research Directions

SAT-Related Computational Problems

- Compute Backbone set, Backdoor set(s), etc. [e.g. Marques-Silva et al, ECAI'10]
- Model Enumeration / Counting
 - Explicit model enumeration
 - Implicit (cube-based) model enumeration [e.g. McMillan, CAV'02]
 - Model counting
- Minimally Unsatisfiable Subformulas
 - Compute one Minimally Unsatisfiable Subformula (MUS)
 - Compute Smallest Minimally Unsatisfiable Subformula (SMUS)
 - Test MUS membership for clause(s)
 - Enumerate all MUSes
- Prime Implicants & Minimal Models
 - Compute one Minimal Model
 - Enumerate all Minimal Models
 - Compute one Prime Implicant
 - Compute Smallest Prime Implicant
 - Enumerate all Prime Implicants

Outline

SAT-Related Computational Problems

Model Enumeration

Minimally Unsatisfiable Subformulas

Prime Implicant & Minimal Models

Extensions of SAT

Example Applications

Research Directions

Model Enumeration & Model Counting

- Model Counting:

- Query: How many models are there?
- Applications: Bayesian networks; etc.
- Techniques:
 - ▶ Connected components
 - ▶ Component caching

[Bayardo & Pehoushek'00]

[Sang et al.'04]

- Model Enumeration:

- Query: List models (represented as cubes)
- Application: Quantification in model checking, etc.
- Techniques:
 - ▶ Clause learning to block computed models

[McMillan'02]

Outline

SAT-Related Computational Problems

Model Enumeration

Minimally Unsatisfiable Subformulas

Prime Implicant & Minimal Models

Extensions of SAT

Example Applications

Research Directions

Resolution Proofs

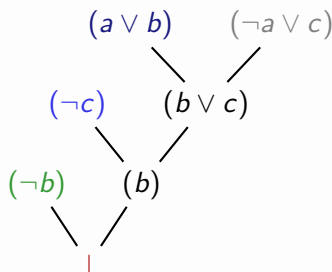
- Resolution operation:

Given $(x \vee \alpha)$ and $(\neg x \vee \beta)$ derive $(\alpha \vee \beta)$

- Consider the formula:

$$\varphi = (\neg c) \wedge (\neg b) \wedge (\neg a \vee c) \wedge (a \vee b) \wedge (a \vee \neg d) \wedge (\neg a \vee \neg d)$$

- Resolution proof:



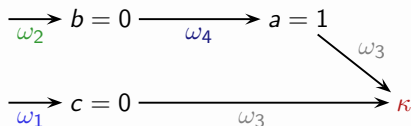
- A modern SAT solver can generate resolution proofs using clauses learnt by the solver

[Zhang&Malik'03]

An Example

- CNF formula:

$$\begin{aligned}\varphi &= \omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 \wedge \omega_5 \wedge \omega_6 \\ &= (\neg c) \wedge (\neg b) \wedge (\neg a \vee c) \wedge (a \vee b) \wedge (a \vee \neg d) \wedge (\neg a \vee \neg d)\end{aligned}$$

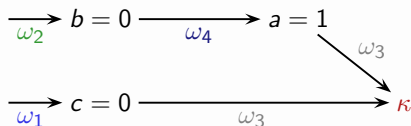


Implication graph with **conflict**

An Example

- CNF formula:

$$\begin{aligned}\varphi &= \omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 \wedge \omega_5 \wedge \omega_6 \\ &= (\neg c) \wedge (\neg b) \wedge (\neg a \vee c) \wedge (a \vee b) \wedge (a \vee \neg d) \wedge (\neg a \vee \neg d)\end{aligned}$$

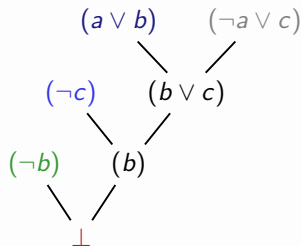
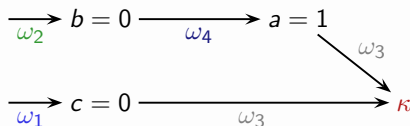


Proof trace $\perp$: $\omega_3 \ \omega_4 \ \omega_1 \ \omega_2$

An Example

- CNF formula:

$$\begin{aligned}\varphi &= \omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 \wedge \omega_5 \wedge \omega_6 \\ &= (\neg c) \wedge (\neg b) \wedge (\neg a \vee c) \wedge (a \vee b) \wedge (a \vee \neg d) \wedge (\neg a \vee \neg d)\end{aligned}$$

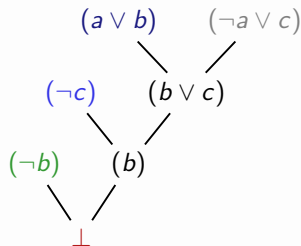
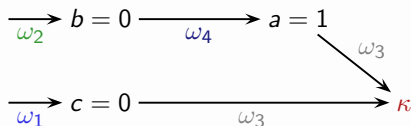


Resolution proof follows **structure of conflicts**

An Example

- CNF formula:

$$\begin{aligned}\varphi &= \omega_1 \wedge \omega_2 \wedge \omega_3 \wedge \omega_4 \wedge \omega_5 \wedge \omega_6 \\ &= (\neg c) \wedge (\neg b) \wedge (\neg a \vee c) \wedge (a \vee b) \wedge (a \vee \neg d) \wedge (\neg a \vee \neg d)\end{aligned}$$



Unsatisfiable subformula: $\omega_1, \omega_2, \omega_3, \omega_4$

Minimally Unsatisfiable Subformulas

- Minimally Unsatisfiable Subformula (MUS)
 $\mathcal{F}$ is unsatisfiable and **any** subset of $\mathcal{F}$ is satisfiable
- Computed unsatisfiable subformulas **approximate** MUSes
- Local search can approximate MUSes
- How to know an MUS approximation is **close to MUS** ?
 - Compute MUSes **exactly** !

Deletion-Based MUS Computation

Input : Unsatisfiable CNF Formula $\mathcal{F}$

Output: MUS $\mathcal{M}$

begin

$\mathcal{M} \leftarrow \mathcal{F}$ // MUS over-approximation

foreach $c \in \mathcal{M}$ **do**

if not SAT($\mathcal{M} \setminus \{c\}$) **then**

$\mathcal{M} \leftarrow \mathcal{M} \setminus \{c\}$ // If UNSAT($\mathcal{M} \setminus \{c\}$), then $c \notin \mathcal{M}$

return $\mathcal{M}$ // Final $\mathcal{M}$ is MUS

end

- Number of calls to SAT oracle: $\Theta(|\mathcal{F}|)$

Deletion-Based MUS Computation

Input : Unsatisfiable CNF Formula $\mathcal{F}$

Output: MUS $\mathcal{M}$

begin

$\mathcal{M} \leftarrow \mathcal{F}$ // MUS over-approximation

foreach $c \in \mathcal{M}$ **do**

if not SAT($\mathcal{M} \setminus \{c\}$) **then**

$\mathcal{M} \leftarrow \mathcal{M} \setminus \{c\}$ // Remove c from $\mathcal{M}$

return $\mathcal{M}$ // Final $\mathcal{M}$ is MUS

end

- Number of calls to SAT oracle: $\Theta(|\mathcal{F}|)$

An Example

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

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

$$(x_1 \vee x_2)$$

$$(\neg x_3)$$

$$(\neg x_2)$$

UNSAT instance

An Example

$(\neg x_1 \vee x_2)$

$(\neg x_3 \vee x_2)$

$(x_1 \vee x_2)$

$(\neg x_3)$

$(\neg x_2)$

Hide clause $(\neg x_1 \vee x_2)$

An Example

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

$$(x_1 \vee x_2)$$

$$(\neg x_3)$$

$$(\neg x_2)$$

SAT instance $\rightarrow$ keep clause $(\neg x_1 \vee x_2)$

An Example

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

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

$$(x_1 \vee x_2)$$

$$(\neg x_3)$$

$$(\neg x_2)$$

Hide clause $(\neg x_3 \vee x_2)$

An Example

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

$$(x_1 \vee x_2)$$

$$(\neg x_3)$$

$$(\neg x_2)$$

UNSAT instance $\rightarrow$ remove clause $(\neg x_3 \vee x_2)$

An Example

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

$$(x_1 \vee x_2)$$

$$(\neg x_3)$$

$$(\neg x_2)$$

Hide clause $(x_1 \vee x_2)$

An Example

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

$$(\neg x_3)$$

$$(\neg x_2)$$

SAT instance $\rightarrow$ keep clause $(x_1 \vee x_2)$

An Example

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

$$(x_1 \vee x_2)$$

$$(\neg x_3)$$

$$(\neg x_2)$$

Hide clause $(\neg x_3)$

An Example

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

$$(x_1 \vee x_2)$$

$$(\neg x_2)$$

UNSAT instance $\rightarrow$ remove clause $(\neg x_3)$

An Example

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

$$(x_1 \vee x_2)$$

$$(\neg x_2)$$

Hide clause $(\neg x_2)$

An Example

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

$$(x_1 \vee x_2)$$

SAT instance $\rightarrow$ keep clause $(\neg x_2)$

An Example

$$\begin{array}{l}(\neg x_1 \vee x_2) \\(x_1 \vee x_2) \\(\neg x_2)\end{array}$$

Computed MUS

An Example

$$\begin{array}{l}(\neg x_1 \vee x_2) \\(x_1 \vee x_2) \\(\neg x_2)\end{array}$$

Computed MUS

- Other MUS algorithms: insertion-based, binary search based, MaxSAT-based, etc.

[e.g. Marques-Silva'10]

Outline

SAT-Related Computational Problems

Model Enumeration

Minimally Unsatisfiable Subformulas

Prime Implicant & Minimal Models

Extensions of SAT

Example Applications

Research Directions

Computing a Prime Implicant

- $\mathcal{F}$ represented in CNF
- Compute satisfying model ν (with SAT solver)
 - ν : represented with true literals of satisfying model
 - $\nu \subsetneq \{x \mid x \in X\} \cup \{\neg x \mid x \in X\}$
- $\mathcal{T}(c, \nu) = c \cap \nu \neq \emptyset$: true literals in clause $c \in \mathcal{F}$ given ν
- $\mathcal{T}(\mathcal{F}, \nu) = \bigcup_{c \in \mathcal{F}} \mathcal{T}(c, \nu) \subseteq \nu$: true literals in all clauses

Computing a Prime Implicant

- $\mathcal{F}$ represented in CNF
- Compute satisfying model ν (with SAT solver)
 - ν : represented with true literals of satisfying model
 - $\nu \subsetneq \{x \mid x \in X\} \cup \{\neg x \mid x \in X\}$
- $\mathcal{T}(c, \nu) = c \cap \nu \neq \emptyset$: true literals in clause $c \in \mathcal{F}$ given ν
- $\mathcal{T}(\mathcal{F}, \nu) = \bigcup_{c \in \mathcal{F}} \mathcal{T}(c, \nu) \subseteq \nu$: true literals in all clauses
- Prime implicant is obtained from:

[e.g. Ravi&Somenzi'04]

$$\begin{aligned} \min \quad & \sum_{I \in \mathcal{T}(\mathcal{F}, \nu)} I \\ \text{s.t.} \quad & \bigwedge_{c \in \mathcal{F}} \left(\bigvee_{I \in \mathcal{T}(c, \nu)} I \right) \end{aligned}$$

- Instance of set/unate covering
 - Solve with PBO or MaxSAT solver

Computing the Smallest Prime Implicant

- $\mathcal{F}$ represented in CNF
- Create set of variables Y from initial set X
 - Construct $\mathcal{F}^Y$ from $\mathcal{F}$
 - ▶ Replace x_i with y_{2i} and $\neg x_i$ with y_{2i+1}

Computing the Smallest Prime Implicant

- $\mathcal{F}$ represented in CNF
- Create set of variables Y from initial set X
 - Construct $\mathcal{F}^Y$ from $\mathcal{F}$
 - ▶ Replace x_i with y_{2i} and $\neg x_i$ with y_{2i+1}
- The **smallest prime implicant** is obtained from:

[e.g. Pizzuti et al.'99]

$$\begin{aligned} \min \quad & \sum_{y \in Y} y \\ \text{s.t.} \quad & \bigwedge_{c \in \mathcal{F}^Y} (c) \\ & \bigwedge_{1 \leq i \leq 2|x|-1} (\neg y_i \vee \neg y_{i+1}) \end{aligned}$$

- Solve with PBO or MaxSAT solver

An Example

- $\mathcal{F} = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$
- The resulting model is:

$$\begin{array}{ll}\min & y_1 + y_2 + y_3 + y_4 \\ \text{s.t.} & (y_1 \vee y_4) \wedge (y_1 \vee y_3) \\ & (\neg y_1 \vee \neg y_2) \wedge (\neg y_3 \vee \neg y_4)\end{array}$$

An Example

- $\mathcal{F} = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$
- The resulting model is:

$$\begin{array}{ll}\min & y_1 + y_2 + y_3 + y_4 \\ \text{s.t.} & (y_1 \vee y_4) \wedge (y_1 \vee y_3) \\ & (\neg y_1 \vee \neg y_2) \wedge (\neg y_3 \vee \neg y_4)\end{array}$$

- The solution is $y_1 = 1, y_2 = y_3 = y_4 = 0$ represents (x_1)

Minimal Models

- $\mathcal{F}$ represented in CNF
- Compute satisfying model ν (with SAT solver)
 - $\nu : X \rightarrow \{0, 1\}$
- $\mathcal{T}(X, \nu) = \{x \in X : \nu(x) = 1\}$: true variables given ν

Minimal Models

- $\mathcal{F}$ represented in CNF
- Compute satisfying model ν (with SAT solver)
 - $\nu : X \rightarrow \{0, 1\}$
- $\mathcal{T}(X, \nu) = \{x \in X : \nu(x) = 1\}$: true variables given ν
- A minimal model (given ν) is obtained from:

$$\begin{array}{ll}\min & \sum_{x \in \mathcal{T}(X, \nu)} x \\ \text{s.t.} & \bigwedge_{c \in \mathcal{F}} (c) \\ & \bigwedge_{x \in X - \mathcal{T}(X, \nu)} (\neg x)\end{array}$$

- Solve with PBO or MaxSAT solver

Outline

SAT-Related Computational Problems

Extensions of SAT

- Pseudo-Boolean Optimization

- Maximum Satisfiability

Example Applications

Research Directions

Well-Known Extensions of SAT

- Quantify the variables
 - Quantified Boolean Formulas (QBF):
Boolean formulas where variables are existentially or universally quantified
- Maximize number of satisfied clauses
 - (Weighted) (Partial) Maximum Satisfiability
- Consider extended constraints
 - Pseudo-Boolean formulas (PBS/PBO):
Linear inequalities over Boolean variables w/o or w/ cost function
- Consider decidable fragments of FOL
 - Satisfiability Modulo Theories (SMT)
Decision procedures for a number of theories exist
 - ▶ Linear Integer Arithmetic; Difference Logic; Uninterpreted Functions; ...
- Other extensions: CSP, ASP
- Some extensions promising; still far from the impact of SAT solvers

Outline

SAT-Related Computational Problems

Extensions of SAT

Pseudo-Boolean Optimization

Maximum Satisfiability

Example Applications

Research Directions

Pseudo-Boolean Constraints & Optimization

- Pseudo-Boolean Constraints:

- Boolean variables: $x_1, \dots, x_n$
- Linear inequalities:

$$\sum_{j \in N} a_{ij} l_j \geq b_i, \quad l_j \in \{x_j, \bar{x}_j\}, x_j \in \{0, 1\}, a_{ij}, b_i, c_j \in \mathbb{N}_0^+$$

- Algorithms:

- ▶ Adapt SAT solver and handle constraints natively
- ▶ Encode constraints to clausal form

[e.g. Manquinho&Marques-Silva'00]

[e.g. Een&Sorensson'06]

- Pseudo-Boolean Optimization:

$$\begin{array}{ll} \text{minimize} & \sum_{j \in N} c_j \cdot x_j \\ \text{subject to} & \sum_{j \in N} a_{ij} l_j \geq b_i, \\ & l_j \in \{x_j, \bar{x}_j\}, x_j \in \{0, 1\}, a_{ij}, b_i, c_j \in \mathbb{N}_0^+ \end{array}$$

Overview of PBO Algorithms

- Iterative upper bound refinement

[e.g. Barth'95, Aloul et al.'02]

- Let B be an upper bound on $\sum_{j \in N} c_j \cdot x_j$
- Create constraint: $\sum_{j \in N} c_j \cdot x_j \leq B - 1$
- Solve with PB solver
- If SAT, upper bound updated to $B - 1$
- Repeat while SAT

Overview of PBO Algorithms

- Iterative upper bound refinement

[e.g. Barth'95, Aloul et al.'02]

- Let B be an upper bound on $\sum_{j \in N} c_j \cdot x_j$
- Create constraint: $\sum_{j \in N} c_j \cdot x_j \leq B - 1$
- Solve with PB solver
- If SAT, upper bound updated to $B - 1$
- Repeat while SAT

- Branch-and-bound search

[e.g. Manquinho&Marques-Silva'00]

- Maintain upper bound (UB) of optimum solution
- Compute estimates of lower bound (LB)
- Bound search if $LB \geq UB$

Outline

SAT-Related Computational Problems

Extensions of SAT

Pseudo-Boolean Optimization

Maximum Satisfiability

Example Applications

Research Directions

What is Maximum Satisfiability?

- CNF Formula:

$$x_6 \vee x_2$$

$$\neg x_6 \vee x_2$$

$$\neg x_2 \vee x_1$$

$$\neg x_1$$

$$\neg x_6 \vee x_8$$

$$x_6 \vee \neg x_8$$

$$x_2 \vee x_4$$

$$\neg x_4 \vee x_5$$

$$x_7 \vee x_5$$

$$\neg x_7 \vee x_5$$

$$\neg x_5 \vee x_3$$

$$\neg x_3$$

What is Maximum Satisfiability?

- CNF Formula:

$$x_6 \vee x_2$$

$$\neg x_6 \vee x_2$$

$$\neg x_6 \vee x_8$$

$$x_6 \vee \neg x_8$$

$$x_7 \vee x_5$$

$$\neg x_7 \vee x_5$$

$$\neg x_2 \vee x_1$$

$$\neg x_1$$

$$x_2 \vee x_4$$

$$\neg x_4 \vee x_5$$

$$\neg x_5 \vee x_3$$

$$\neg x_3$$

- Formula is **unsatisfiable**
- **MaxSAT**:
 - Find assignment that **maximizes** number of satisfied clauses
 - ▶ For above formula, solution is **10**
- Can be used for solving (linear) **Boolean optimization** problems
- There are a number of variants of **MaxSAT**

The MaxSAT Problem(s)

- MaxSAT:
 - All clauses are **soft**
 - Find assignment that maximizes number of satisfied **soft** clauses
- Partial MaxSAT:
 - Hard clauses **must** be satisfied
 - Find assignment that maximizes number of satisfied **soft** clauses
- Weighted MaxSAT
 - Weights associated with clauses
 - Find assignment that maximizes sum of weights of satisfied clauses
- Weighted partial MaxSAT
 - Weights associated with **soft** clauses
 - Hard clauses **must** be satisfied
 - Find assignment that maximizes sum of weights of satisfied **soft** clauses

Representative MaxSAT Solutions

- Branch and bound search
 - Sophisticated **lower bounds** (on # of unsatisfiable clauses)
 - ▶ Unit propagation [Li, Manya & Planes'05]
 - ▶ Inconsistent subsets [Li, Manya & Planes'06]
 - Dedicated **inference techniques**
 - ▶ Adapted resolution rule [Heras & Larrosa'06]
 - ▶ ...
- **Cannot** use most effective SAT techniques
 - **No** unit propagation
 - **No** unrestricted clause learning
 - ...

Solving MaxSAT with PB I

[E.g. Amgoud et al.'96, Aloul et al.'02]

- Formula φ with n variables and $m = |\varphi|$ clauses
- Create φ' from φ :
 - Replace each clause ω_i with $\omega'_i = \omega_i \cup \{b_i\}$
 - One new **relaxation** (or **blocking**) variable b_i for each clause
 - φ' with m clauses and $n + m$ variables
- Trivial to satisfy φ' by assigning $b_i = 1$, for all i
- Example:
 - CNF formula φ :

$$\varphi = \{\{x_1, \neg x_2\}, \{x_1, x_2\}, \{\neg x_1\}\}$$

- Modified formula φ' :

$$\varphi' = \{\{x_1, \neg x_2, b_1\}, \{x_1, x_2, b_2\}, \{\neg x_1, b_3\}\}$$

Solving MaxSAT with PB II

- Iteratively solve the modified PB/SAT problem:

$$\varphi'' = \varphi' \cup \left\{ \sum b_i \leq k \right\}$$

- Start with $k = m - 1$
- Decrease value of k while φ'' is satisfiable
- Stop when φ'' becomes unsatisfiable
 - ▶ Return $k + 1$ (i.e. value of k for last satisfiable φ'')
- Example:
 - Recall modified formula $\varphi' = \{ \{x_1, \neg x_2, b_1\}, \{x_1, x_2, b_2\}, \{\neg x_1, b_3\} \}$
 - Instances of PB/SAT to solve, w/ $k = 2, 1, 0$:

$$\varphi'' = \{ \{x_1, \neg x_2, b_1\}, \{x_1, x_2, b_2\}, \{\neg x_1, b_3\} \} \cup \left\{ \sum b_i \leq k \right\}$$

- ▶ Satisfiable for $k = 2, 1$; Unsatisfiable for $k = 0$
- ▶ Return $k = 1$

Solving MaxSAT with PB III

- Can use PB solver or SAT solver
 - With SAT, just encode cardinality constraint $\sum b_i \leq k$ into CNF
- Can **refine** cardinality constraint and **avoid** redoing search
- Can **reduce** number of iterations
 - E.g. update k given number of b_i variables assigned value 1
- Does **not** scale in practice (from MaxSAT evaluation 2008):

Class	# Instances	PBO	
		MaxSAT	minisat+
DEBUG	65	63	–
FIR	59	–	37
SYN	74	–	44
MTG	215	–	0

Solving MaxSAT with PB IV

- Why does MaxSAT with PB not scale ?
 - Number of variables is now $n + |\varphi|$
 - Complex cardinality constraints
 - ▶ E.g. $n = 10^5$ and $m = 10^6$: $\sum b_i \leq 10^6 - 1$
- Use of PB might be feasible if number of b_i 's could be reduced
 - Can we be selective on which clauses to add b_i variables to ?

Solving MaxSAT with PB IV

- Why does MaxSAT with PB not scale ?
 - Number of variables is now $n + |\varphi|$
 - Complex cardinality constraints
 - ▶ E.g. $n = 10^5$ and $m = 10^6$: $\sum b_i \leq 10^6 - 1$
- Use of PB might be feasible if number of b_i 's could be reduced
 - Can we be selective on which clauses to add b_i variables to ?
 - ▶ **Yes !** Only need to **relax clauses in unsatisfiable cores**
 - ▶ Need to identify clauses in **unsatisfiable cores**

Many Unsatisfiability-Based MaxSAT Algorithms

- (Partial) MaxSAT:

[Fu&Malik'06, Marques-Silva&Planes'08, Marques-Silva&Manquinho'08, Ansotegui et al.'09, etc.]

- msu1.X ($X = 0, 1, 2$) ; msu2.0 ; msu3.Y ($Y = 0, 1$) ; msu4.0 ; pm2

- Weighted (Partial) MaxSAT: [Manquinho et al.'09, Ansotegui et al.'09, Ansotegui et al.'10]

- wmsu1 / wpm1 ; wpm2

- Main differences:

- Number of cardinality constraints
- CNF encoding of cardinality constraints
- Progression of the algorithm
 - ▶ Refine LB vs. refine LB&UB
 - ▶ All unsatisfiable instances vs. some possibly satisfiable
- Single relaxation variable per clause vs. multiple relaxation variables

An Example: msu1.0

$$x_6 \vee x_2$$

$$\neg x_6 \vee x_2$$

$$\neg x_2 \vee x_1$$

$$\neg x_1$$

$$\neg x_6 \vee x_8$$

$$x_6 \vee \neg x_8$$

$$x_2 \vee x_4$$

$$\neg x_4 \vee x_5$$

$$x_7 \vee x_5$$

$$\neg x_7 \vee x_5$$

$$\neg x_5 \vee x_3$$

$$\neg x_3$$

Example CNF formula

An Example: msu1.0

$$x_6 \vee x_2 \qquad \neg x_6 \vee x_2$$

$$\neg x_6 \vee x_8 \qquad x_6 \vee \neg x_8$$

$$x_7 \vee x_5 \qquad \neg x_7 \vee x_5$$

$$\neg x_2 \vee x_1$$

$$\neg x_1$$

$$x_2 \vee x_4$$

$$\neg x_4 \vee x_5$$

$$\neg x_5 \vee x_3$$

$$\neg x_3$$

Formula is **UNSAT**; Get unsat core

An Example: msu1.0

$$x_6 \vee x_2$$

$$\neg x_6 \vee x_2$$

$$\neg x_2 \vee x_1 \vee b_1$$

$$\neg x_1 \vee b_2$$

$$\neg x_6 \vee x_8$$

$$x_6 \vee \neg x_8$$

$$x_2 \vee x_4 \vee b_3$$

$$\neg x_4 \vee x_5 \vee b_4$$

$$x_7 \vee x_5$$

$$\neg x_7 \vee x_5$$

$$\neg x_5 \vee x_3 \vee b_5$$

$$\neg x_3 \vee b_6$$

$$\sum_{i=1}^6 b_i \leq 1$$

Add **relaxation variables** and AtMost1 constraint

An Example: msu1.0

$$x_6 \vee x_2$$

$$\neg x_6 \vee x_2$$

$$\neg x_2 \vee x_1 \vee b_1$$

$$\neg x_1 \vee b_2$$

$$\neg x_6 \vee x_8$$

$$x_6 \vee \neg x_8$$

$$x_2 \vee x_4 \vee b_3$$

$$\neg x_4 \vee x_5 \vee b_4$$

$$x_7 \vee x_5$$

$$\neg x_7 \vee x_5$$

$$\neg x_5 \vee x_3 \vee b_5$$

$$\neg x_3 \vee b_6$$

$$\sum_{i=1}^6 b_i \leq 1$$

Formula is (again) **UNSAT**; Get unsat core

An Example: msu1.0

$$x_6 \vee x_2 \vee b_7 \quad \neg x_6 \vee x_2 \vee b_8 \quad \neg x_2 \vee x_1 \vee b_1 \vee b_9 \quad \neg x_1 \vee b_2 \vee b_{10}$$

$$\neg x_6 \vee x_8 \quad x_6 \vee \neg x_8 \quad x_2 \vee x_4 \vee b_3 \quad \neg x_4 \vee x_5 \vee b_4$$

$$x_7 \vee x_5 \vee b_{11} \quad \neg x_7 \vee x_5 \vee b_{12} \quad \neg x_5 \vee x_3 \vee b_5 \vee b_{13} \quad \neg x_3 \vee b_6 \vee b_{14}$$

$$\sum_{i=1}^6 b_i \leq 1 \quad \sum_{i=7}^{14} b_i \leq 1$$

Add new relaxation variables and AtMost1 constraint

An Example: msu1.0

$$x_6 \vee x_2 \vee b_7 \quad \neg x_6 \vee x_2 \vee b_8 \quad \neg x_2 \vee x_1 \vee b_1 \vee b_9 \quad \neg x_1 \vee b_2 \vee b_{10}$$

$$\neg x_6 \vee x_8 \quad x_6 \vee \neg x_8 \quad x_2 \vee x_4 \vee b_3 \quad \neg x_4 \vee x_5 \vee b_4$$

$$x_7 \vee x_5 \vee b_{11} \quad \neg x_7 \vee x_5 \vee b_{12} \quad \neg x_5 \vee x_3 \vee b_5 \vee b_{13} \quad \neg x_3 \vee b_6 \vee b_{14}$$

$$\sum_{i=1}^6 b_i \leq 1 \quad \sum_{i=7}^{14} b_i \leq 1$$

Instance is now SAT

An Example: msu1.0

$$x_6 \vee x_2 \vee b_7 \quad \neg x_6 \vee x_2 \vee b_8 \quad \neg x_2 \vee x_1 \vee b_1 \vee b_9 \quad \neg x_1 \vee b_2 \vee b_{10}$$

$$\neg x_6 \vee x_8 \quad x_6 \vee \neg x_8 \quad x_2 \vee x_4 \vee b_3 \quad \neg x_4 \vee x_5 \vee b_4$$

$$x_7 \vee x_5 \vee b_{11} \quad \neg x_7 \vee x_5 \vee b_{12} \quad \neg x_5 \vee x_3 \vee b_5 \vee b_{13} \quad \neg x_3 \vee b_6 \vee b_{14}$$

$$\sum_{i=1}^6 b_i \leq 1 \quad \sum_{i=7}^{14} b_i \leq 1$$

MaxSAT solution is $|\varphi| - \mathcal{I} = 12 - 2 = 10$

Organization of msu1.0

[Fu&Malik'06]

- Clauses characterized as:
 - **Initial**: derived from clauses in φ
 - **Auxiliary**: added during execution of algorithm
 - ▶ E.g. clauses from cardinality constraints
- While exist unsatisfiable cores
 - Add **fresh** set B of relaxation variables to **non-auxiliary soft** clauses in core
 - Add **new** AtMost1 constraint

$$\sum_{b_i \in B} b_i \leq 1$$

- ▶ At most 1 relaxation variable from set B can take value 1

- MaxSAT solution is $|\varphi| - \mathcal{I}$, where $\mathcal{I}$ is number of iterations

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

- Graph Coloring

- Minimum Vertex Cover

- N-Queens

- Software Installability

- Design Debugging

- Equivalence Checking

- Haplotype Inference

- ATPG

- Bounded Model Checking

Research Directions

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

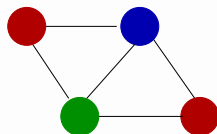
ATPG

Bounded Model Checking

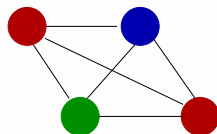
Research Directions

Example: Graph Coloring I

- Decide whether one can assign one of K colors to each of the vertices of graph $G = (V, E)$ such that adjacent vertices are assigned different colors



Valid coloring



Invalid coloring

Example: Graph Coloring II

- Given $N = |V|$ vertices and K colors, create $N \times K$ variables:
 $x_{ij} = 1$ iff vertex i is assigned color j ; 0 otherwise
- For each edge (u, v) , require different assigned colors to u and v :

$$1 \leq j \leq K, \quad (\neg x_{uj} \vee \neg x_{vj})$$

- Each vertex is assigned exactly one color:

$$1 \leq i \leq N, \quad \sum_{j=1}^K x_{ij} = 1$$

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

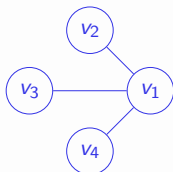
ATPG

Bounded Model Checking

Research Directions

Minimum Vertex Cover I

- The problem:
 - Graph $G = (V, E)$
 - Vertex cover $U \subseteq V$, such that for all edges (v_i, v_j) , either $v_i \in U$ or $v_j \in U$.
 - Minimum vertex cover: vertex cover U of minimum size



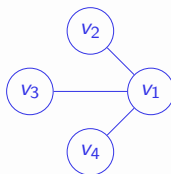
Min vertex cover: $\{v_1\}$

Minimum Vertex Cover II

- Partial MaxSAT formulation:
 - Associate x_i with each $v_i \in V$, such that $x_i = 1$ iff $v_i \in U$, otherwise $v_i \in V - U$.
 - **Hard** clauses: $(x_i \vee x_j)$ for each edge $(v_i, v_j) \in E$
 - **Soft** clauses: $(\neg x_i)$ for each vertex v_i
 - ▶ I.e. preferable **not** to include vertices in U

Minimum Vertex Cover II

- Partial MaxSAT formulation:
 - Associate x_i with each $v_i \in V$, such that $x_i = 1$ iff $v_i \in U$, otherwise $v_i \in V - U$.
 - **Hard** clauses: $(x_i \vee x_j)$ for each edge $(v_i, v_j) \in E$
 - **Soft** clauses: $(\neg x_i)$ for each vertex v_i
 - ▶ I.e. preferable **not** to include vertices in U



$$\varphi_H = \{(x_1 \vee x_2), (x_1 \vee x_3), (x_1 \vee x_4)\}$$

$$\varphi_S = \{(\neg x_1), (\neg x_2), (\neg x_3), (\neg x_4)\}$$

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

ATPG

Bounded Model Checking

Research Directions

Example: The N-Queens Problem I

- The N-Queens Problem:
Place N queens on a $N \times N$ board, such that no two queens attack each other
- Example for a 5×5 board:

Q				
			Q	
	Q			
				Q
		Q		

Example: The N-Queens Problem II

- x_{ij} : 1 if queen placed in position (i, j) ; 0 otherwise
- Each row must have exactly one queen:

$$1 \leq i \leq N, \quad \sum_{j=1}^N x_{ij} = 1$$

- Each column must have exactly one queen:

$$1 \leq j \leq N, \quad \sum_{i=1}^N x_{ij} = 1$$

- Also, need to define constraints on diagonals...

Example: The N-Queens Problem III

- Each diagonal can have at most one queen:

↘	↗	↗	↗	
↘				↖
↘				↖
↘				↖
↗	↗	↗	↗	

$$i = 1, \quad 2 \leq j < N, \quad \sum_{k=0}^{j-1} x_{i+k, j-k} \leq 1$$

$$i = N, \quad 1 \leq j < N, \quad \sum_{k=0}^{N-j} x_{i-k, j+k} \leq 1$$

$$j = 1, \quad 1 \leq i < N, \quad \sum_{k=0}^{N-i} x_{i+k, j+k} \leq 1$$

$$j = N, \quad 2 \leq i < N, \quad \sum_{k=0}^{i-1} x_{i-k, j-k} \leq 1$$

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

ATPG

Bounded Model Checking

Research Directions

Software Installability Problem

[Argelich&Lynce'08]

- Set of software packages: $\{p_1, \dots, p_n\}$
- Associate x_i with p_i : $x_i = 1$ iff p_i is installed
- Constraints associated with package p_i : (p_i, D_i, C_i)
 - D_i : dependencies for installing p_i
 - C_i : conflicts for installing p_i
- Example problem: **Maximum Installability Problem**
 - Maximum number of packages that can be installed
 - Package constraints represent **hard** clauses
 - **Soft** clauses: (x_i)

Package constraints:

$(p_1, \{p_2 \vee p_3\}, \{p_4\})$
 $(p_2, \{p_3\}, \{p_4\})$
 $(p_3, \{p_2\}, \emptyset)$
 $(p_4, \{p_2, p_3\}, \emptyset)$

MaxSAT formulation:

$\varphi_H = \{(\neg x_1 \vee x_2 \vee x_3), (\neg x_1 \vee \neg x_4),$
 $(\neg x_2 \vee x_3), (\neg x_2 \vee \neg x_4), (\neg x_3 \vee x_2),$
 $(\neg x_4 \vee x_2), (\neg x_4 \vee x_3)\}$
 $\varphi_S = \{(x_1), (x_2), (x_3), (x_4)\}$

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

ATPG

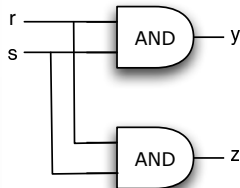
Bounded Model Checking

Research Directions

Design Debugging

[Safarpour et al.'07]

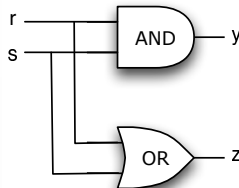
Correct circuit



Input stimuli: $\langle r, s \rangle = \langle 0, 1 \rangle$

Valid output: $\langle y, z \rangle = \langle 0, 0 \rangle$

Faulty circuit



Input stimuli: $\langle r, s \rangle = \langle 0, 1 \rangle$

Invalid output: $\langle y, z \rangle = \langle 0, 0 \rangle$

- The model:
 - **Hard** clauses: Input and output values
 - **Soft** clauses: CNF representation of circuit
- The problem:
 - Maximize number of satisfied clauses (i.e. circuit gates)

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

ATPG

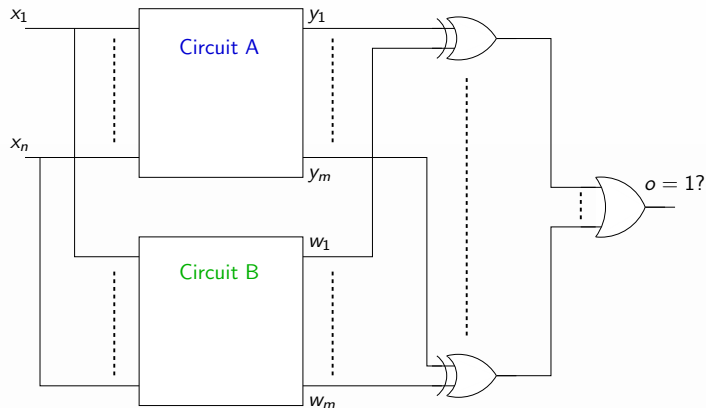
Bounded Model Checking

Research Directions

The CEC Problem

- Combinational circuit C_A , with n inputs and m outputs
- Combinational circuit C_B , with n inputs and m outputs
- The **Combinational Equivalence Checking (CEC)** Problem?
 - Are the two circuits equivalent?
 - A complicated problem formulation:
 - ▶ Are the outputs equivalent for **all** input values?
 - A simpler alternative:
 - ▶ Are there (at least one) input values that distinguish outputs of the two circuits?

Solving CEC with a Miter



- Can easily be represented in CNF, and solved with SAT solver
 - **Note:** CEC is a difficult problem, and more sophisticated techniques are often used

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

ATPG

Bounded Model Checking

Research Directions

Haplotype Inference by Pure Parsimony (HIPP)

- Assume a set G of n strings over the alphabet $\{0, 1, 2\}$ (the **genotypes**), each with m characters
 - Each character j in a string g_i represented by g_{ij}
 - Example: $g_i = 012$
- A string g_i is **explained** by two strings over the alphabet $\{0, 1\}$ (the **haplotypes**), h_a and h_b , iff:
 - If $g_{ij} = 0$, then $h_{aj} = h_{bj} = 0$
 - If $g_{ij} = 1$, then $h_{aj} = h_{bj} = 1$
 - If $g_{ij} = 2$, then $h_{aj} \neq h_{bj}$
 - Example: $g_i = 012$ is explained by $h_a = 010$ and $h_b = 011$
- The HIPP problem is to compute a **minimum-size** set H of **haplotypes** such that every **genotype** in G is explained by two haplotypes in H
 - The HIPP problem is NP-Hard

An Example of the HIPP Problem ($n = 18$ and $m = 5$)

00000
00010
00020
01101
01102
01200
02000
02020
02200
02202
02220
02222
11000
21000
21200
21202
22000
22020

An Example of the HIPP Problem ($n = 18$ and $m = 5$)

00000	=	00000	$\oplus$	00000
00010	=	00010	$\oplus$	00010
00020	=	00000	$\oplus$	00010
01101	=	01101	$\oplus$	01101
01102	=	01100	$\oplus$	01101
01200	=	01100	$\oplus$	01000
02000	=	00000	$\oplus$	01000
02020	=	01000	$\oplus$	00010
02200	=	00000	$\oplus$	01100
02202	=	00000	$\oplus$	01101
02220	=	00010	$\oplus$	01100
02222	=	00010	$\oplus$	01101
11000	=	11000	$\oplus$	11000
21000	=	11000	$\oplus$	01000
21200	=	01100	$\oplus$	11000
21202	=	01101	$\oplus$	11000
22000	=	11000	$\oplus$	00000
22020	=	00010	$\oplus$	11000

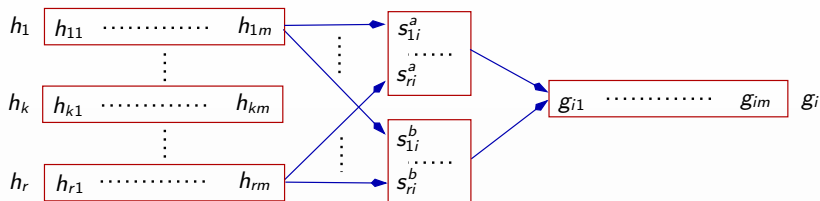
- HIPP solution has size 6

SHIPs: SAT-based Haplotype Inference by Pure Parsimony

[Lynce&Marques-Silva'06]

- Clearly, the number of haplotypes lies between 1 and $2n$,
 $1 \leq |H| \leq 2n$
 - It is in general possible to find tighter lower and upper bounds on the number of haplotypes
- SHIPs iteratively considers increasing numbers of candidate haplotypes
 - Start from lower bound lb
 - ▶ A trivial value is 1, but can compute better lower bounds
 - Terminate for a value of r when all genotypes can be explained by r haplotypes
 - ▶ Guaranteed to terminate until $r = 2n$

Model for r Candidate Haplotypes I



- Must **select** two haplotypes for explaining each genotype g_i
- g variables only needed for sites with value 2
- Candidate haplotypes represented with $r \times m$ Boolean variables (h)
- Selector variables represented with $2 \times r \times n$ Boolean variables (s)

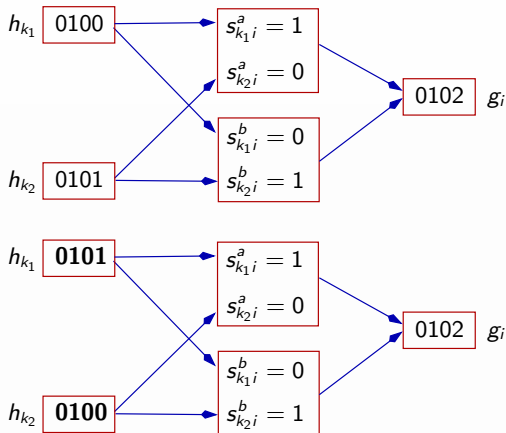
Model for r Candidate Haplotypes II

- Conditions on sites: (with $1 \leq k \leq r$)
 - If $g_{ij} = 0$, add constraints $(s_{ki}^a \rightarrow \neg h_{kj})$ and $(s_{ki}^b \rightarrow \neg h_{kj})$
 - ▶ If h_k is used for explaining g_i , then $h_{kj} = g_{ij} = 0$
 - If $g_{ij} = 1$, add constraints $(s_{ki}^a \rightarrow h_{kj})$ and $(s_{ki}^b \rightarrow h_{kj})$
 - ▶ If h_k is used for explaining g_i , then $h_{kj} = g_{ij} = 1$
 - If $g_{ij} = 2$:
 - ▶ Add variable t_{ij}
 - $t_{ij} = 0$ if g_{ij} explained by 01 pair
 - $t_{ij} = 1$ if g_{ij} explained by 10 pair
 - ▶ Add clauses relating h and t variables:

$$s_{ki}^a \rightarrow (h_{kj} \leftrightarrow t_{ij})$$
$$s_{ki}^b \rightarrow \neg(h_{kj} \leftrightarrow t_{ij})$$

Model for r Candidate Haplotypes III

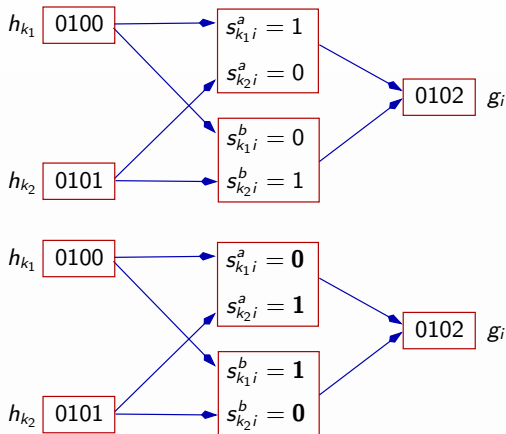
- Problem formulation has **key** symmetries on the h variables:



- Enforce an ordering of the Boolean valuations to the haplotypes
 - Require $h_1^v < h_2^v < \dots < h_r^v$ for any valuation v

Model for r Candidate Haplotypes IV

- Problem formulation also has **key** symmetries on the **s** variables:



Model for r Candidate Haplotypes V

- Symmetries on s variables can be eliminated:
 - Haplotype selected by s_{ki}^a variables must have index **smaller** than haplotype selected by s_{ki}^b variables; with $k_2 < k_1$:

$$\left(s_{k_1 i}^a \rightarrow \bigwedge_{k_2=1}^{k_1-1} \neg s_{k_2 i}^b \right) \text{ and } \left(s_{k_2 i}^b \rightarrow \bigwedge_{k_1=k_2+1}^r \neg s_{k_1 i}^a \right)$$

Model for r Candidate Haplotypes V

- Conditions on selector variables:
 - Exactly one haplotype (for a and for b) is selected for each genotype g_i :

$$\left(\sum_{k=1}^r s_{ki}^a = 1 \right) \wedge \left(\sum_{k=1}^r s_{ki}^b = 1 \right)$$

- Can be represented with clauses linear in r
- Space complexity of the model:
 - Worst-case ($r = \Theta(n)$): $\mathcal{O}(n^2 \times m)$
 - In practice ($r = \mathcal{O}(n)$): $\mathcal{O}(n \times r \times m)$
 - ▶ In practice SAT model significantly more compact than existing ILP models

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

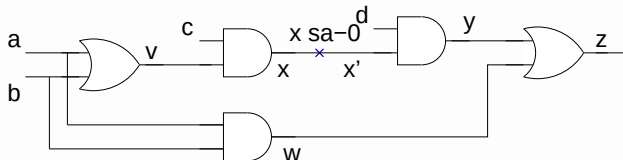
ATPG

Bounded Model Checking

Research Directions

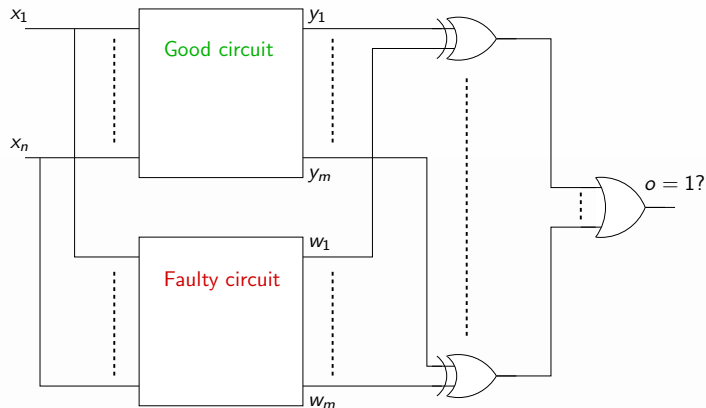
The ATPG Problem

- Digital systems can exhibit **defects**
- Physical defects are modeled as **logical faults**
- Most often used fault model: **single stuck-at fault model**
 - Circuit lines **stuck-at** a fixed logic value
 - ▶ **sa-0**: Fixed at 0
 - ▶ **sa-1**: Fixed at 1
- Example:



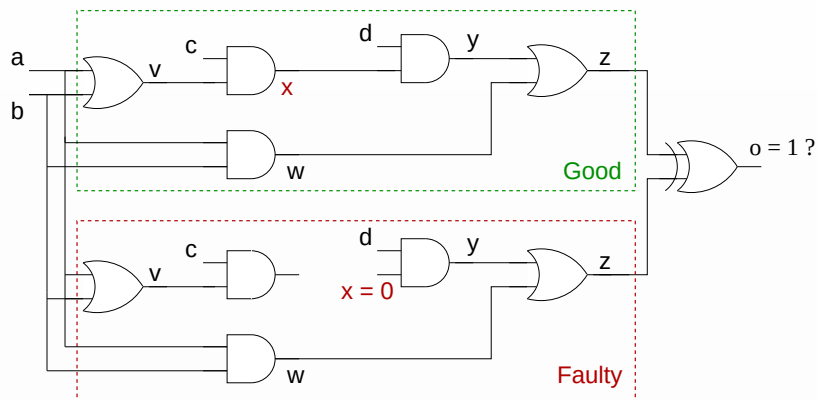
- Fault *x* stuck-at 0 causes line *x* to be fixed at value 0

Solving ATPG with a Miter



- Can easily be represented in CNF, and solved with SAT solver
 - **Note 1:** Simplistic model; actual SAT formulation significantly more complex
 - **Note 2:** Dedicated algorithms can be competitive with SAT

Example Miter for ATPG



Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

Graph Coloring

Minimum Vertex Cover

N-Queens

Software Installability

Design Debugging

Equivalence Checking

Haplotype Inference

ATPG

Bounded Model Checking

Research Directions

Unfolding the Transition Relation

[Biere et al.'99]

- Hardware and software systems abstracted as **state transition systems**
- Initial state predicate: $I(s_0)$
- Transition relation represented by predicate: $T(s_i, s_{i+1})$
- Unfold transition relation for k time frames starting from initial state:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1})$$

- Interpretation:
 - Create k **copies** of transition relation predicate, requiring one of the initial states

Representing Safety Properties

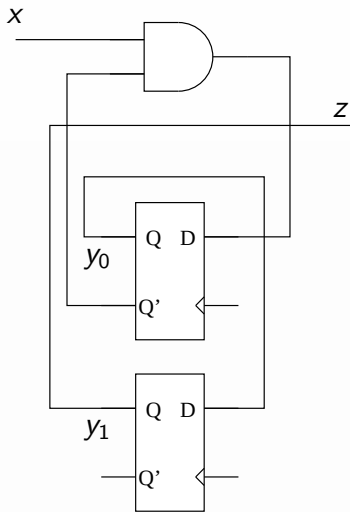
- **Safety** property: $AG\ p$
 - **Meaning**: property p holds **globally** in **all** computation paths
 - To use SAT negate property, i.e. $EF\ \neg p$ and check for $\neg p$
 - Check property for each unfolded computation step, i.e. p_i
- Initial state predicate: $I(s_0)$
- Transition relation represented by predicate: $T(s_i, s_{i+1})$
- SAT formulation:

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \left(\bigvee_{i=0}^{i=k} \neg p_i \right)$$

Bounded Model Checking (BMC)

- Start from given unfolding size B
- While unfolding less than user-specified bound
 - Generate SAT formula for target unfolding size
 - Invoke SAT solver
 - If formula is SAT return counterexample
 - Unfold further
- BMC is in general incomplete
 - Can find bugs and provide counterexamples
 - May be unable to prove that properties hold
 - Completeness achieved if length of reachability diameter is known
 - ▶ **Reachability diameter**: longest shortest path in transition system

An Example I



- Define next state equations:

$$y'_0 = x \wedge \bar{y}_0$$

$$y'_1 = y_0$$

- Define transition relation:

$$- s_i = y_{1,i} y_{0,i}$$

$$- s_{i+1} = y_{1,i+1} y_{0,i+1}$$

$$T(s_i, s_{i+1}) = (y_{0,i+1} \leftrightarrow (x_i \wedge \bar{y}_{0,i})) \wedge (y_{1,i+1} \leftrightarrow y_{0,i})$$

- Can represent $T(s_i, s_{i+1})$ in CNF

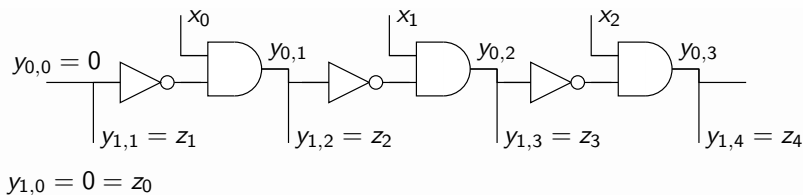
- Initial state predicate:

$$I(s_0) = (\neg y_{0,0}) \wedge (\neg y_{1,0})$$

- Example temporal property: $AG(\neg y_1)$

An Example II

- Unfolding considered: 3 computation steps
- Easy to represent in CNF



- Property considered $(y_{1,0} \vee y_{1,1} \vee y_{1,2} \vee y_{1,3})$
- Counterexample: enough to set $x_0 = 1$
 - Which sets $y_{1,2} = 1$
- $EF(y_1)$ holds; hence property $AG(\neg y_1)$ does **not** hold

Outline

SAT-Related Computational Problems

Extensions of SAT

Example Applications

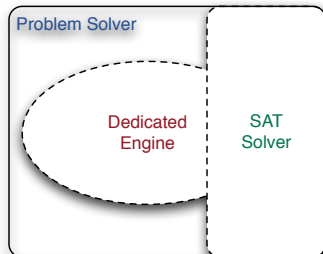
Research Directions

Domain Extensions

- Use richer modelling languages & allow for optimization
 - Pseudo-Boolean Constraints (PB)
 - ▶ Cardinality constraints, general PB constraints, etc.
 - ▶ Simple non-linear constraints
 - (Partial) (Weighted) MaxSAT
 - ▶ Soft clauses/constraints
 - Answer Set Programming (ASP)
 - Quantified Boolean Formulas (QBF)
 - ▶ Variable quantification
 - Satisfiability Modulo Theories (SMT)
 - ▶ Decidable fragments of FOL
 - Constraint Programming (?)
 - First Order Logic (?)

SAT-Based Problem Solving

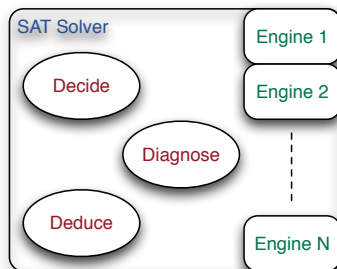
- More Applications
 - Hundreds (thousands?) of documented applications of SAT and extensions of SAT
 - (Many?) more to be expected
- Better Modelling
 - Many SAT applications; some with **naive** SAT modelling
 - ▶ Sophisticated SAT-based modelling
- SAT solvers as **oracles**, **but**:
 - White-box SAT solver integration
 - ▶ Also for SAT extensions



- Access to computed models
- Access to unsatisfiable subformulas
- Access to learned clauses
- Access to variable/clause activity information
- ...

Problem-Specific Solvers

- Embed dedicated solvers within SAT solver
 - Adapt DPLL(T) framework used in SMT



- Concrete examples:
 - ▶ Handling parity constraints
 - ▶ (Some) PBO solvers

[Laitinen et al., ECAI'10]

[Manquinho&Marques-Silva, AMAI'04]

Algorithmic Improvements

- Emulate extended resolution
 - Add/learn new variables/definitions, in addition to clause learning

[Audemard et al., AAAI'10; Huang, AIJ'10]

- Integrate Stalmarck's Dilemma rule
 - Probe $UP(a)$ and $UP(\neg a)$
 - Compute intersection

[Sheeran&Stalmarck, FMCAD'98]

- Integrate recursive learning's rule
 - Probe $UP(I)$, for $I \in \omega$
 - Compute intersection

[Kunz&Pradhan, TCAD'94]

Exploit Tractability

- Enumerate models of polynomially-solvable subproblems
 - E.g. enumerate 2CNFSAT models [Larrabee, TCAD'92]
 - Impractical in general: **number of models can be exponential**

Exploit Tractability

- Enumerate models of polynomially-solvable subproblems
 - E.g. enumerate 2CNFSAT models [Larrabee, TCAD'92]
 - Impractical in general: **number of models can be exponential**
- Solve subproblems to find lower bounds in optimization problems
 - Applicable to PBO, MaxSAT, MaxSMT, etc.
 - Solve polynomial subclasses
 - ▶ E.g. 2CNFSAT, Horn-SAT, etc.
 - ▶ Does not capture **hard** part of problem instances
 - Recent advances in fixed-parameter algorithms
 - ▶ E.g. MaxSAT
 - ▶ Compute lower bounds by solving (extended) subproblems

Thank you