Encoding Redundancy for Satisfaction-Driven Clause Learning

Marijn J.H. Heule

Benjamin Kiesl

Armin Biere

THE UNIVERSITY OF TEXAS AT AUSTIN

CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

JOHANNES KEPLER UNIVERSITÄT LINZ

The Problem

Although SAT solvers can often handle gigantic formulas, they sometimes fail miserably on seemingly easy problems.

Outline

Outline

SAT Solving in Practice: Gigantic Search Trees

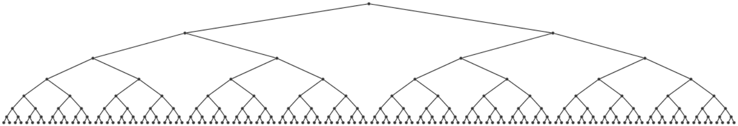
- SAT: Given a propositional formula, is it satisfiable?
  - Formulas usually in CNF: $(x \vee y) \wedge (\bar{x} \vee \bar{y}) \wedge (z \vee \bar{z})$

SAT Solving in Practice: Gigantic Search Trees

- SAT: Given a propositional formula, is it satisfiable?
  - Formulas usually in CNF: $(x \vee y) \wedge (\bar{x} \vee \bar{y}) \wedge (z \vee \bar{z})$
- Prototypical NP-complete problem.
  - ➥ No known algorithm for SAT that runs in polynomial time.

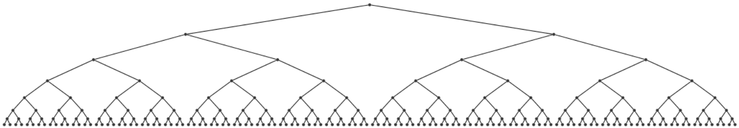
SAT Solving in Practice: Gigantic Search Trees

- SAT: Given a propositional formula, is it satisfiable?
  - Formulas usually in CNF: $(x \vee y) \wedge (\bar{x} \vee \bar{y}) \wedge (z \vee \bar{z})$
- Prototypical NP-complete problem.
  - ➥ No known algorithm for SAT that runs in polynomial time.
- Search tree for only seven variables (leaves $\Leftrightarrow$ assignments):



SAT Solving in Practice: Gigantic Search Trees

- SAT: Given a propositional formula, is it satisfiable?
  - Formulas usually in CNF: $(x \vee y) \wedge (\bar{x} \vee \bar{y}) \wedge (z \vee \bar{z})$
- Prototypical NP-complete problem.
  - ➥ No known algorithm for SAT that runs in polynomial time.
- Search tree for only seven variables (leaves $\Leftrightarrow$ assignments):



- Modern solvers often deal with millions of variables and clauses.

SAT: Problem Solved?

- Pigeonhole Principle: If n pigeons are put into $n-1$ holes, then at least one hole must contain two pigeons.

SAT: Problem Solved?

- Pigeonhole Principle: If n pigeons are put into $n - 1$ holes, then at least one hole must contain two pigeons.

- My little nephew could figure this out.

SAT: Problem Solved?

- Pigeonhole Principle: If n pigeons are put into $n-1$ holes, then at least one hole must contain two pigeons.

- My little nephew could figure this out.
- What if we encode it into SAT and pass it to a solver?

SAT Solver: 21 Pigeons Into 20 Holes?

SAT Solver: 21 Pigeons Into 20 Holes?

"Arguably the single most studied combinatorial principle in all of proof complexity." [Nordström, SIGLOG News '15]

Seemingly Easy, Awfully Hard: Not Only the Pigeons

- There exist many seemingly easy formulas that are awfully hard for modern SAT solvers.
    - Formulas are often unsatisfiable ($\Rightarrow$ co-NP).

Seemingly Easy, Awfully Hard: Not Only the Pigeons

- There exist many seemingly easy formulas that are awfully hard for modern SAT solvers.
  - Formulas are often unsatisfiable ($\Rightarrow$ co-NP).
- Proof complexity can explain why some of them are so hard:
  - Some formulas have only resolution proofs of exponential size.

Seemingly Easy, Awfully Hard: Not Only the Pigeons

- There exist many seemingly easy formulas that are awfully hard for modern SAT solvers.
    - Formulas are often unsatisfiable ($\Rightarrow$ co-NP).
- Proof complexity can explain why some of them are so hard:
    - Some formulas have only resolution proofs of exponential size.
    - Modern solvers are usually based on Conflict-Driven Clause Learning (CDCL), which is based on the resolution proof system.

Seemingly Easy, Awfully Hard: Not Only the Pigeons

- There exist many seemingly easy formulas that are awfully hard for modern SAT solvers.
  - Formulas are often unsatisfiable ($\Rightarrow$ co-NP).
- Proof complexity can explain why some of them are so hard:
  - Some formulas have only resolution proofs of exponential size.
  - Modern solvers are usually based on Conflict-Driven Clause Learning (CDCL), which is based on the resolution proof system.
  - CDCL solvers basically construct a resolution proof during solving.

Seemingly Easy, Awfully Hard: Not Only the Pigeons

- There exist many seemingly easy formulas that are awfully hard for modern SAT solvers.
  - Formulas are often unsatisfiable ($\Rightarrow$ co-NP).
- Proof complexity can explain why some of them are so hard:
  - Some formulas have only resolution proofs of exponential size.
  - Modern solvers are usually based on Conflict-Driven Clause Learning (CDCL), which is based on the resolution proof system.
  - CDCL solvers basically construct a resolution proof during solving.

➥ They need exponential time to solve these formulas.

There is no Easy Way

- No matter how much engineering effort we put into a CDCL solver, it will never be able solve the hard formulas!

There is no Easy Way

- No matter how much engineering effort we put into a CDCL solver, it will never be able solve the hard formulas!
  - The exponential gap stems from an inherent theoretical restriction.

There is no Easy Way

- No matter how much engineering effort we put into a CDCL solver, it will never be able solve the hard formulas!
  - The exponential gap stems from an inherent theoretical restriction.

- What is needed to jump over this gap?

There is no Easy Way

- No matter how much engineering effort we put into a CDCL solver, it will never be able solve the hard formulas!
  - The exponential gap stems from an inherent theoretical restriction.

- What is needed to jump over this gap?
  1. a proof system that is stronger than resolution yet still mechanizable: PR proof system [Heule, K, Biere; CADE '17].

There is no Easy Way

- No matter how much engineering effort we put into a CDCL solver, it will never be able solve the hard formulas!
  - The exponential gap stems from an inherent theoretical restriction.

- What is needed to jump over this gap?
  1. a proof system that is stronger than resolution yet still mechanizable: PR proof system [Heule, K, Biere; CADE '17].
  2. a SAT solving paradigm harnessing the strength of PR: satisfaction-driven clause learning [Heule, K, Seidl, Biere; HVC '17].

Satisfaction-Driven Clause Learning (SDCL): General Idea

- CDCL learns clauses that are implied.

Satisfaction-Driven Clause Learning (SDCL): General Idea

- CDCL learns clauses that are implied.
- SDCL only requires learned clauses to be redundant (not implied):

Definition

A clause C is redundant with respect to a formula F if F and $F \wedge C$ are equisatisfiable.

Satisfaction-Driven Clause Learning (SDCL): General Idea

- CDCL learns clauses that are implied.
- SDCL only requires learned clauses to be redundant (not implied):

Definition

A clause C is redundant with respect to a formula F if F and $F \wedge C$ are equisatisfiable.

- Only allow clauses that fulfill an efficiently decidable redundancy criterion: propagation redundancy (PR) [Heule, K, Biere; CADE '17]
  - "mother of all efficiently decidable redundancy criteria".

Satisfaction-Driven Clause Learning (SDCL): General Idea

- CDCL learns clauses that are implied.
- SDCL only requires learned clauses to be redundant (not implied):

Definition

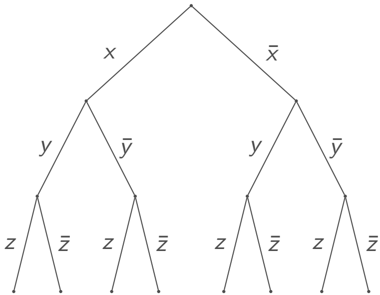
A clause C is redundant with respect to a formula F if F and $F \wedge C$ are equisatisfiable.

- Only allow clauses that fulfill an efficiently decidable redundancy criterion: propagation redundancy (PR) [Heule, K, Biere; CADE '17]
  - "mother of all efficiently decidable redundancy criteria".

➥ Addition of redundant clauses can prune the search tree.

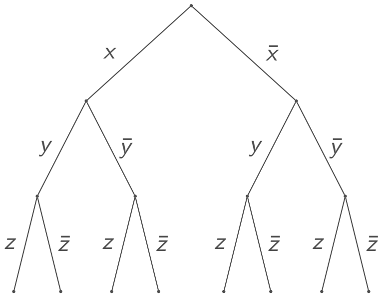
Clause Addition ↔ Pruning

- Clause addition prunes the search tree of satisfying assignments.



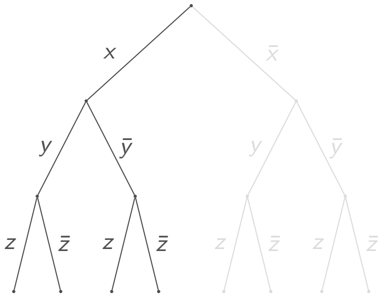
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.



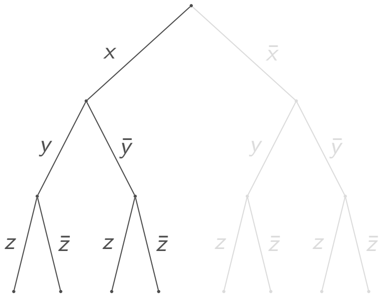
Clause Addition ↔ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.



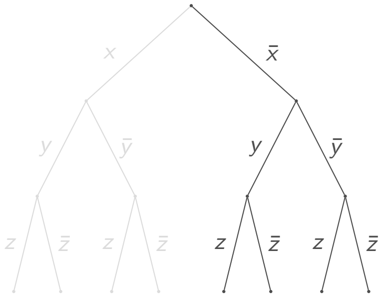
Clause Addition ↔ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples:



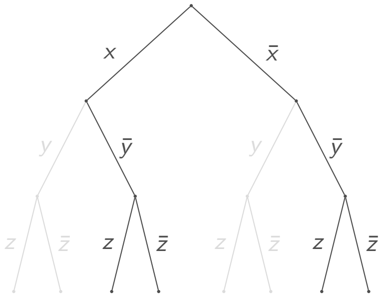
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$



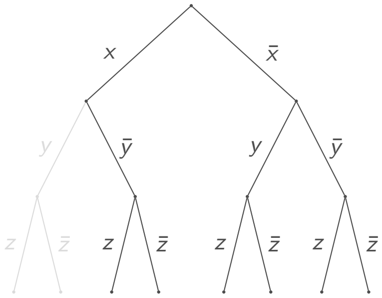
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$



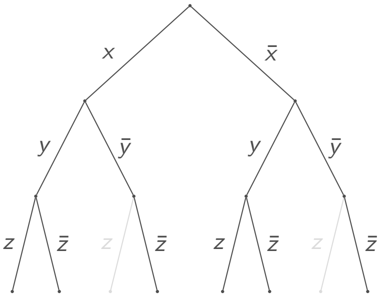
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$



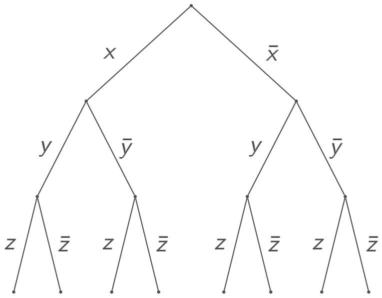
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$



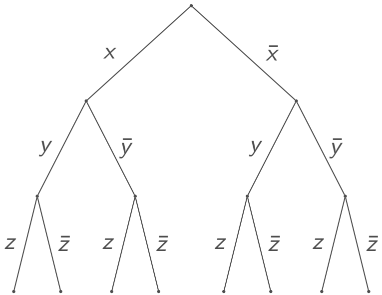
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$ $(x \vee \bar{x})$



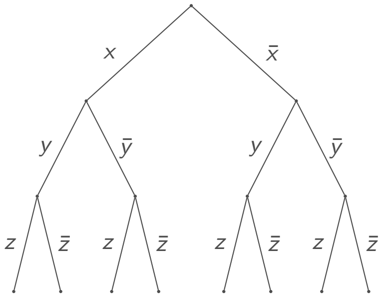
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$ $(x \vee \bar{x})$
- Addition of multiple clauses combines all the "clause prunings".



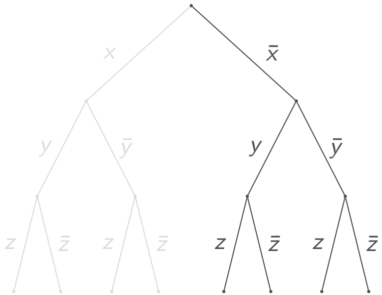
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$ $(x \vee \bar{x})$
- Addition of multiple clauses combines all the "clause prunings".
- Example:



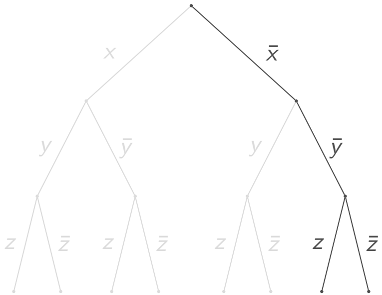
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$ $(x \vee \bar{x})$
- Addition of multiple clauses combines all the "clause prunings".
- Example: $(\bar{x})$



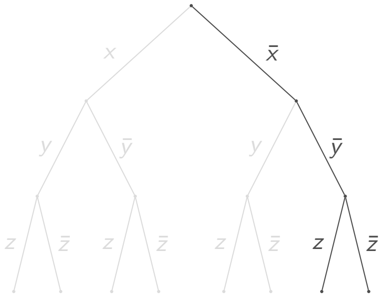
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$ $(x \vee \bar{x})$
- Addition of multiple clauses combines all the "clause prunings".
- Example: $(\bar{x}) \wedge (\bar{y})$



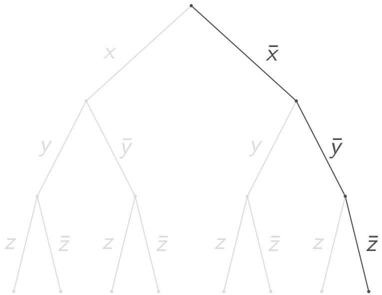
Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$ $(x \vee \bar{x})$
- Addition of multiple clauses combines all the "clause prunings".
- Example: $(\bar{x}) \wedge (\bar{y}) \wedge (\bar{x} \vee \bar{y})$



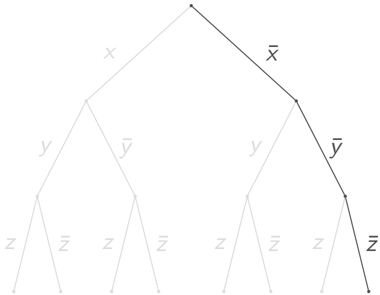
Clause Addition ↔ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$ $(x \vee \bar{x})$
- Addition of multiple clauses combines all the "clause prunings".
- Example: $(\bar{x}) \wedge (\bar{y}) \wedge (\bar{x} \vee \bar{y}) \wedge (y \vee \bar{z})$



Clause Addition $\leftrightarrow$ Pruning

- Clause addition prunes the search tree of satisfying assignments.
- Example: The clause (x) prunes all branches where x is false.
- Other Examples: $(\bar{x})$ $(\bar{y})$ $(\bar{x} \vee \bar{y})$ $(y \vee \bar{z})$ $(x \vee \bar{x})$
- Addition of multiple clauses combines all the "clause prunings".
- Example: $(\bar{x}) \wedge (\bar{y}) \wedge (\bar{x} \vee \bar{y}) \wedge (y \vee \bar{z}) \wedge (x \vee \bar{x})$



Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].

Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.

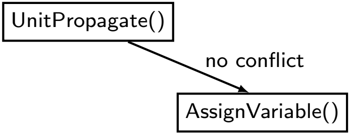
Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.

UnitPropagate()

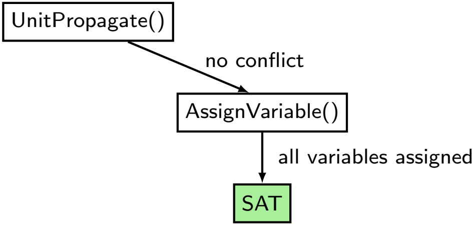
Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.



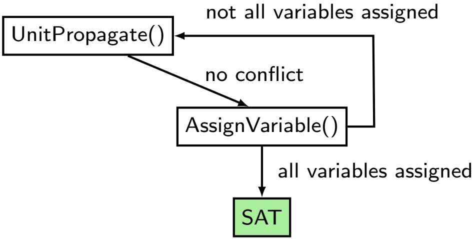
Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.



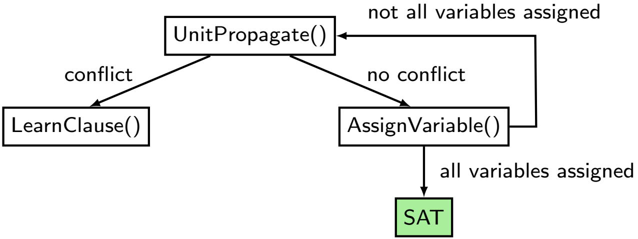
Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.



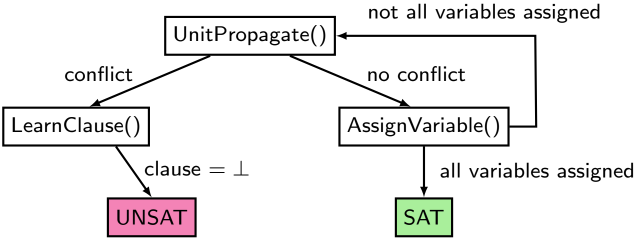
Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.



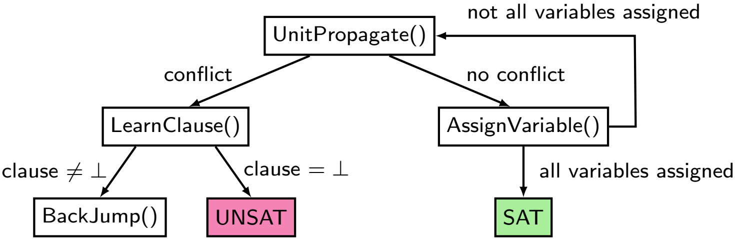
Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.



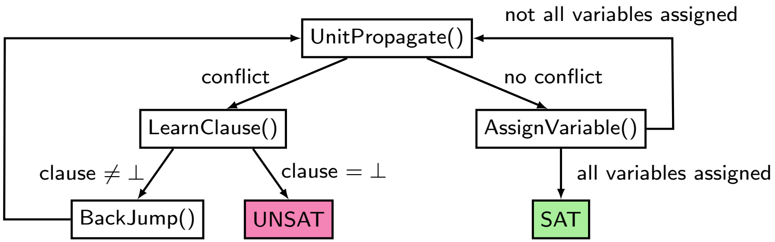
Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.



Conflict-Driven Clause Learning (CDCL)

- By Marques-Silva and Sakallah [ICCAD '96] as well as Moskewicz, Madigan, Zhao, Zhang, and Malik [DAC '01].
- Key ideas:
  - Simplify the formula with unit propagation; then assign a variable. Repeat until the formula is solved.
  - Learn clauses to avoid "bad" assignments in the future.

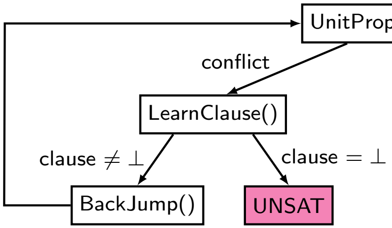


Satisfaction-Driven Clause Learning (SDCL)

- Key idea: if unit propagation does not derive a conflict, try to prune (part of) the current assignment from the search tree.

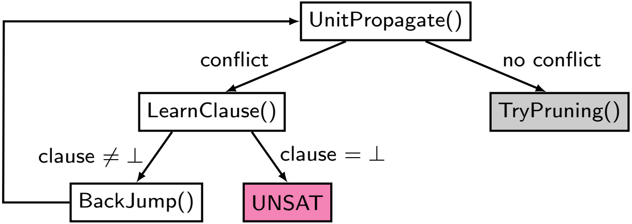
Satisfaction-Driven Clause Learning (SDCL)

- Key idea: if unit propagation does not derive a conflict, try to prune (part of) the current assignment from the search tree.



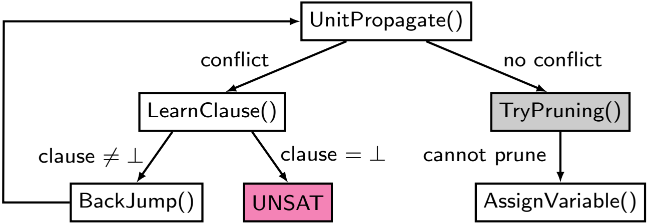
Satisfaction-Driven Clause Learning (SDCL)

- Key idea: if unit propagation does not derive a conflict, try to prune (part of) the current assignment from the search tree.



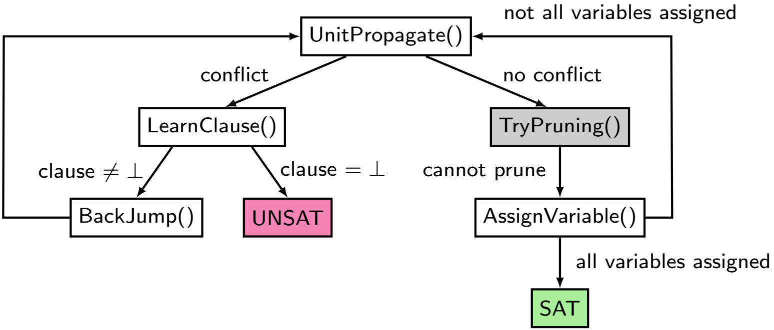
Satisfaction-Driven Clause Learning (SDCL)

- Key idea: if unit propagation does not derive a conflict, try to prune (part of) the current assignment from the search tree.



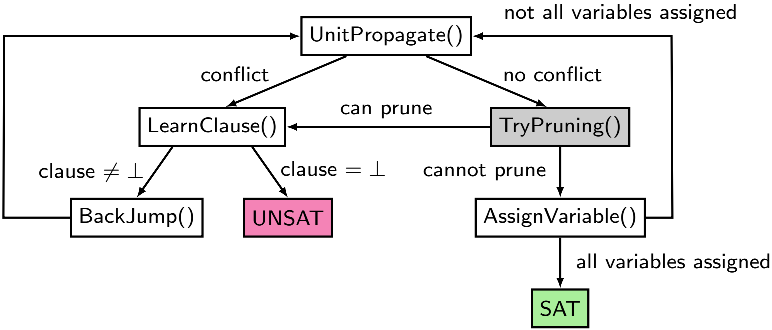
Satisfaction-Driven Clause Learning (SDCL)

- Key idea: if unit propagation does not derive a conflict, try to prune (part of) the current assignment from the search tree.



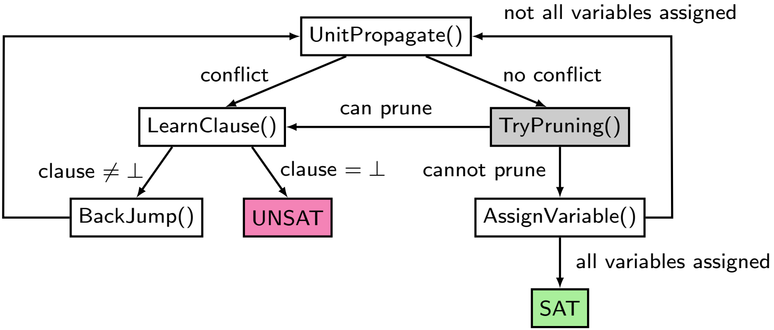
Satisfaction-Driven Clause Learning (SDCL)

- Key idea: if unit propagation does not derive a conflict, try to prune (part of) the current assignment from the search tree.



Satisfaction-Driven Clause Learning (SDCL)

- Key idea: if unit propagation does not derive a conflict, try to prune (part of) the current assignment from the search tree.



- Learned clauses are not necessarily implied (PR clauses).

How to Check if the Search Tree Can be Pruned

- When can we prune? Encode question into SAT!

How to Check if the Search Tree Can be Pruned

- When can we prune? Encode question into SAT!
- Solver produces a simple formula that is satisfiable if the current assignment can be pruned.

How to Check if the Search Tree Can be Pruned

- When can we prune? Encode question into SAT!
- Solver produces a simple formula that is satisfiable if the current assignment can be pruned.
  - Originally ("positive reduct") [Heule, K, Seidl, Biere; HVC '17]: Take all satisfied clauses and remove unassigned literals, then add the clause that is blocked by the current assignment.

How to Check if the Search Tree Can be Pruned

- When can we prune? Encode question into SAT!
- Solver produces a simple formula that is satisfiable if the current assignment can be pruned.
  - Originally ("positive reduct") [Heule, K, Seidl, Biere; HVC '17]: Take all satisfied clauses and remove unassigned literals, then add the clause that is blocked by the current assignment.
- Solver then calls a "child solver" to solve the simpler formula.

How to Check if the Search Tree Can be Pruned

- When can we prune? Encode question into SAT!
- Solver produces a simple formula that is satisfiable if the current assignment can be pruned.
  - Originally ("positive reduct") [Heule, K, Seidl, Biere; HVC '17]: Take all satisfied clauses and remove unassigned literals, then add the clause that is blocked by the current assignment.
- Solver then calls a "child solver" to solve the simpler formula.
- Problem: Positive reduct only works on pigeonhole formulas but not on other hard formulas.

How to Check if the Search Tree Can be Pruned

- When can we prune? Encode question into SAT!
- Solver produces a simple formula that is satisfiable if the current assignment can be pruned.
  - Originally ("positive reduct") [Heule, K, Seidl, Biere; HVC '17]: Take all satisfied clauses and remove unassigned literals, then add the clause that is blocked by the current assignment.
- Solver then calls a "child solver" to solve the simpler formula.
- Problem: Positive reduct only works on pigeonhole formulas but not on other hard formulas.

➥ Wanted: Better encodings for pruning!

Background

Contribution

Encodings for Stronger Pruning: Some Preliminaries

- $F|_\alpha$ denotes the application of the assignment α to F (remove all clauses satisfied by α and then remove all literals falsified by α)

Encodings for Stronger Pruning: Some Preliminaries

- $F|_\alpha$ denotes the application of the assignment α to F (remove all clauses satisfied by α and then remove all literals falsified by α)
- For an assignment $\alpha = a_1 \ldots a_n$, we define $\bar{\alpha} = (\bar{a}_1 \vee \cdots \vee \bar{a}_n)$.

Encodings for Stronger Pruning: Some Preliminaries

- $F|_\alpha$ denotes the application of the assignment α to F (remove all clauses satisfied by α and then remove all literals falsified by α)
- For an assignment $\alpha = a_1 \dots a_n$, we define $\bar{\alpha} = (\bar{a}_1 \vee \dots \vee \bar{a}_n)$.
- $\text{touched}_\alpha(C)$ denotes the subclause of C that is assigned by α.

Encodings for Stronger Pruning: Some Preliminaries

- $F|_\alpha$ denotes the application of the assignment α to F (remove all clauses satisfied by α and then remove all literals falsified by α)
- For an assignment $\alpha = a_1 \ldots a_n$, we define $\bar{\alpha} = (\bar{a}_1 \vee \cdots \vee \bar{a}_n)$.
- $\text{touched}_\alpha(C)$ denotes the subclause of C that is assigned by α.
- Notion of implication via unit propagation:
  - Clauses: $F \vdash_1 C$ iff unit propagation derives a conflict on $F \wedge \bar{C}$.
  - Formulas: $F \vdash_1 G$ iff $F \vdash_1 C$ for all $C \in G$.

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

Definition

Let F be a formula and α an assignment. Then, the filtered positive reduct $\mathrm{f}_\alpha(F)$ of F and α is the formula $G \wedge \bar{\alpha}$ where $G = \{\mathrm{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \not\vdash_1 \mathrm{untouched}_\alpha(D)\}$.

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

Definition

Let F be a formula and α an assignment. Then, the filtered positive reduct $\mathsf{f}_\alpha(F)$ of F and α is the formula $G \wedge \bar{\alpha}$ where $G = \{\text{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \nvdash_1 \text{untouched}_\alpha(D)\}$.

Theorem

If the filtered positive reduct $\mathsf{f}_\alpha(F)$ is satisfiable, then F and $F \wedge \bar{\alpha}$ are equisatisfiable.

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

Definition

Let F be a formula and α an assignment. Then, the filtered positive reduct $\mathsf{f}_\alpha(F)$ of F and α is the formula $G \wedge \bar{\alpha}$ where $G = \{\text{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \not\vdash_1 \text{untouched}_\alpha(D)\}$.

Theorem

If the filtered positive reduct $\mathsf{f}_\alpha(F)$ *is satisfiable, then* F *and* $F \wedge \bar{\alpha}$ *are equisatisfiable.*

➥ Works very well in practice (see later)!

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

Definition

Let F be a formula and α an assignment. Then, the filtered positive reduct $\mathrm{f}_\alpha(F)$ of F and α is the formula $G \wedge \bar{\alpha}$ where $G = \{\text{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \nvdash_1 \text{untouched}_\alpha(D)\}$.

- Example: $F = (x \vee y) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee z)$ and $\alpha = x$.

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

Definition

Let F be a formula and α an assignment. Then, the filtered positive reduct $\mathrm{f}_\alpha(F)$ of F and α is the formula $G \wedge \bar{\alpha}$ where $G = \{\mathrm{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \nvdash_1 \mathrm{untouched}_\alpha(D)\}$.

- Example: $F = (x \vee y) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee z)$ and $\alpha = x$.
  - filtered positive reduct $\mathrm{f}_\alpha(F) = (\bar{x})$

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

Definition

Let F be a formula and α an assignment. Then, the filtered positive reduct $\mathsf{f}_\alpha(F)$ of F and α is the formula $G \wedge \bar{\alpha}$ where $G = \{\text{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \nvdash_1 \text{untouched}_\alpha(D)\}$.

- Example: $F = (x \vee y) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee z)$ and $\alpha = x$.
  - filtered positive reduct $\mathsf{f}_\alpha(F) = (\bar{x}) \Rightarrow$ satisfiable $\Rightarrow$ can prune α

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

Definition

Let F be a formula and α an assignment. Then, the filtered positive reduct $\mathsf{f}_\alpha(F)$ of F and α is the formula $G \wedge \bar{\alpha}$ where $G = \{\text{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \nvdash_1 \text{untouched}_\alpha(D)\}$.

- Example: $F = (x \vee y) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee z)$ and $\alpha = x$.
  - filtered positive reduct $\mathsf{f}_\alpha(F) = (\bar{x}) \Rightarrow$ satisfiable $\Rightarrow$ can prune α
  - positive reduct $\mathsf{p}_\alpha(F) = (x) \wedge (\bar{x})$

New Contribution: Filtered Positive Reduct

- The filtered positive reduct is a subset of the positive reduct:

Definition

Let F be a formula and α an assignment. Then, the filtered positive reduct $\mathrm{f}_\alpha(F)$ of F and α is the formula $G \wedge \bar{\alpha}$ where $G = \{\mathrm{touched}_\alpha(D) \mid D \in F \text{ and } F|_\alpha \not\vdash_1 \mathrm{untouched}_\alpha(D)\}$.

- Example: $F = (x \vee y) \wedge (\bar{x} \vee y) \wedge (\bar{y} \vee z)$ and $\alpha = x$.
  - filtered positive reduct $\mathrm{f}_\alpha(F) = (\bar{x}) \Rightarrow$ satisfiable $\Rightarrow$ can prune α
  - positive reduct $\mathrm{p}_\alpha(F) = (x) \wedge (\bar{x}) \Rightarrow$ unsatisfiable $\Rightarrow$ can't prune α

Even Stronger Pruning: PR Reduct

- PR reduct (don't try to understand this):

Definition

Let F be a formula and α an assignment. Then, the PR reduct $\mathrm{pr}_\alpha(F)$ of F and α is the formula $G \wedge C$ where C is the clause that blocks α and G is the union of the following sets of clauses where all the s_i are new variables:

$$\{\bar{x}^p \vee \bar{x}^n \mid x \in var(F) \setminus var(\alpha)\},$$

$$\{\bar{s}_i \vee \mathrm{touched}_\alpha(D_i) \vee \mathrm{untouched}_\alpha(D_i)^p \mid D_i \in F\},$$

$$\{\overline{L^n} \vee s_i \mid D_i \in F \text{ and } L \subseteq \mathrm{untouched}_\alpha(D_i)$$
$$\text{such that } F\,|_\alpha \nvdash_1 \mathrm{untouched}_\alpha(D_i) \setminus L\}.$$

Even Stronger Pruning: PR Reduct (continued)

- Allows for even stronger pruning than the filtered positive reduct.

Even Stronger Pruning: PR Reduct (continued)

- Allows for even stronger pruning than the filtered positive reduct.
- Precisely characterizes propagation redundancy.
  - ➥ Extremely general redundancy notion (NP-hard).

Even Stronger Pruning: PR Reduct (continued)

- Allows for even stronger pruning than the filtered positive reduct.
- Precisely characterizes propagation redundancy.
  - ➥ Extremely general redundancy notion (NP-hard).
- Has other nice theoretical properties.

Even Stronger Pruning: PR Reduct (continued)

- Allows for even stronger pruning than the filtered positive reduct.
- Precisely characterizes propagation redundancy.
  - ➥ Extremely general redundancy notion (NP-hard).
- Has other nice theoretical properties.
- Doesn't work well in practice
  - ➥ Constructing and solving take too long.

Evaluation: SDCL in Practice

- SDCL solver, called SaDiCaL (by Armin Biere).

Evaluation: SDCL in Practice

- SDCL solver, called SaDiCaL (by Armin Biere).
  - implemented from scratch, efficient CDCL part, simple.

Evaluation: SDCL in Practice

- SDCL solver, called SaDiCaL (by Armin Biere).
  - implemented from scratch, efficient CDCL part, simple.
- SaDiCaL can produce short PR proofs of formulas for which CDCL solvers require exponential time:
  - pigeonhole principle,
  - Tseitin formulas over expander graphs, and
  - mutilated chessboard formulas.

Evaluation: SDCL in Practice

- SDCL solver, called SaDiCaL (by Armin Biere).
  - implemented from scratch, efficient CDCL part, simple.
- SaDiCaL can produce short PR proofs of formulas for which CDCL solvers require exponential time:
  - pigeonhole principle,
  - Tseitin formulas over expander graphs, and
  - mutilated chessboard formulas.
- Three of the most popular formula families hard for resolution.

Evaluation: SDCL in Practice

- SDCL solver, called SaDiCaL (by Armin Biere).
  - implemented from scratch, efficient CDCL part, simple.
- SaDiCaL can produce short PR proofs of formulas for which CDCL solvers require exponential time:
  - pigeonhole principle,
  - Tseitin formulas over expander graphs, and
  - mutilated chessboard formulas.
- Three of the most popular formula families hard for resolution.
- Proofs validated by formally verified proof checkers.

Evaluation: SDCL in Practice

- SDCL solver, called SaDiCaL (by Armin Biere).
  - implemented from scratch, efficient CDCL part, simple.
- SaDiCaL can produce short PR proofs of formulas for which CDCL solvers require exponential time:
  - pigeonhole principle,
  - Tseitin formulas over expander graphs, and
  - mutilated chessboard formulas.
- Three of the most popular formula families hard for resolution.
- Proofs validated by formally verified proof checkers.
- Robust w.r.t. scrambling for Tseitin formulas and mutilated chessboards.

Experimental Data: Pigeonhole Principle

Formula	MLBT	Plain	Pos. Red.	F. Red
hole20	> 3600	> 3600	0.26	0.49
hole30	> 3600	> 3600	1.96	4.03
hole40	> 3600	> 3600	9.02	19.54
hole50	> 3600	> 3600	28.63	65.90

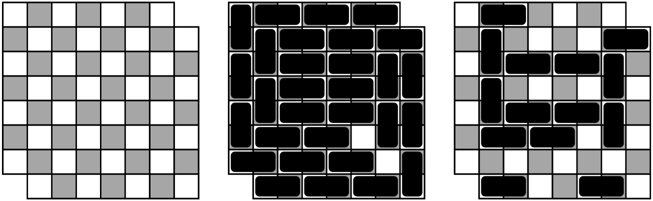
- MLBT – MapleLCMDistChronoBT (winner SAT Competition 2018)
- Plain – SaDiCaL in CDCL mode

Experimental Data: Tseitin Formulas

Formula	MLBT	Plain	Pos. Red.	F. Red
Urquhart-s3-b1	2.95	16.31	> 3600	0.02
Urquhart-s3-b2	1.36	2.82	> 3600	0.03
Urquhart-s3-b3	2.28	2.08	> 3600	0.03
Urquhart-s3-b4	10.74	7.65	> 3600	0.03
Urquhart-s4-b1	86.11	> 3600	> 3600	0.32
Urquhart-s4-b2	154.35	183.77	> 3600	0.11
Urquhart-s4-b3	258.46	129.27	> 3600	0.16
Urquhart-s4-b4	> 3600	> 3600	> 3600	0.14
Urquhart-s5-b1	> 3600	> 3600	> 3600	1.27
Urquhart-s5-b2	> 3600	> 3600	> 3600	0.58
Urquhart-s5-b3	> 3600	> 3600	> 3600	1.67
Urquhart-s5-b4	> 3600	> 3600	> 3600	2.91

Experimental Data: Mutilated Chessboards

Formula	`MLBT`	Plain	Pos. Red.	F. Red
mchess_15	51.53	2480.67	> 3600	13.14
mchess_16	380.45	2115.75	> 3600	15.52
mchess_17	2418.35	> 3600	> 3600	25.54
mchess_18	> 3600	> 3600	> 3600	43.88



Summary

- SAT solving paradigm for hard unsatisfiable formulas: SDCL

Summary

- SAT solving paradigm for hard unsatisfiable formulas: SDCL
- New encodings allow for stronger pruning:

Summary

- SAT solving paradigm for hard unsatisfiable formulas: SDCL
- New encodings allow for stronger pruning:
  - Filtered positive reduct works well in practice.

Summary

- SAT solving paradigm for hard unsatisfiable formulas: SDCL
- New encodings allow for stronger pruning:
  - Filtered positive reduct works well in practice.
  - PR reduct characterizes propagation redundancy but doesn't work well in practice.

Summary

- SAT solving paradigm for hard unsatisfiable formulas: SDCL
- New encodings allow for stronger pruning:
  - Filtered positive reduct works well in practice.
  - PR reduct characterizes propagation redundancy but doesn't work well in practice.
- Solver SaDiCaL produces checkable proofs of formula families that are popular for being extremely hard.

Summary

- SAT solving paradigm for hard unsatisfiable formulas: SDCL
- New encodings allow for stronger pruning:
  - Filtered positive reduct works well in practice.
  - PR reduct characterizes propagation redundancy but doesn't work well in practice.
- Solver SaDiCaL produces checkable proofs of formula families that are popular for being extremely hard.
- Next step: SDCL for hard problems from cryptanalysis?