SAT Solving

Shaowei Cai

Institute of Software, Chinese Academy of Sciences

2017.3.24

Outline

Outline

Outline

Outline

The SAT Problem

- A set of boolean variables: $X = \{x_1, x_2, ..., x_n\}$.
- Literals: $x_1, \neg x_1, x_2, \neg x_2, ..., x_n, \neg x_n$
- A set of Clauses: $x_1 \vee \neg x_2, x_2 \vee x_3, x_2 \vee \neg x_4, \neg x_1 \vee \neg x_3 \vee x_4, ...$
- A Conjunctive Normal Form (CNF) formula:

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

The SAT Problem

- A set of boolean variables: $X = \{x_1, x_2, ..., x_n\}$.
- Literals: $x_1, \neg x_1, x_2, \neg x_2, ..., x_n, \neg x_n$
- A set of Clauses: $x_1 \vee \neg x_2, x_2 \vee x_3, x_2 \vee \neg x_4, \neg x_1 \vee \neg x_3 \vee x_4, ...$
- A Conjunctive Normal Form (CNF) formula:

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

- The satisfiability problem (SAT): test whether there exists an assignment of truth values to the variables in F under which F evaluates to true.

About SAT

- The first NP-Complete problem, starting NP-complete theory [Stephen Cook, 1971]

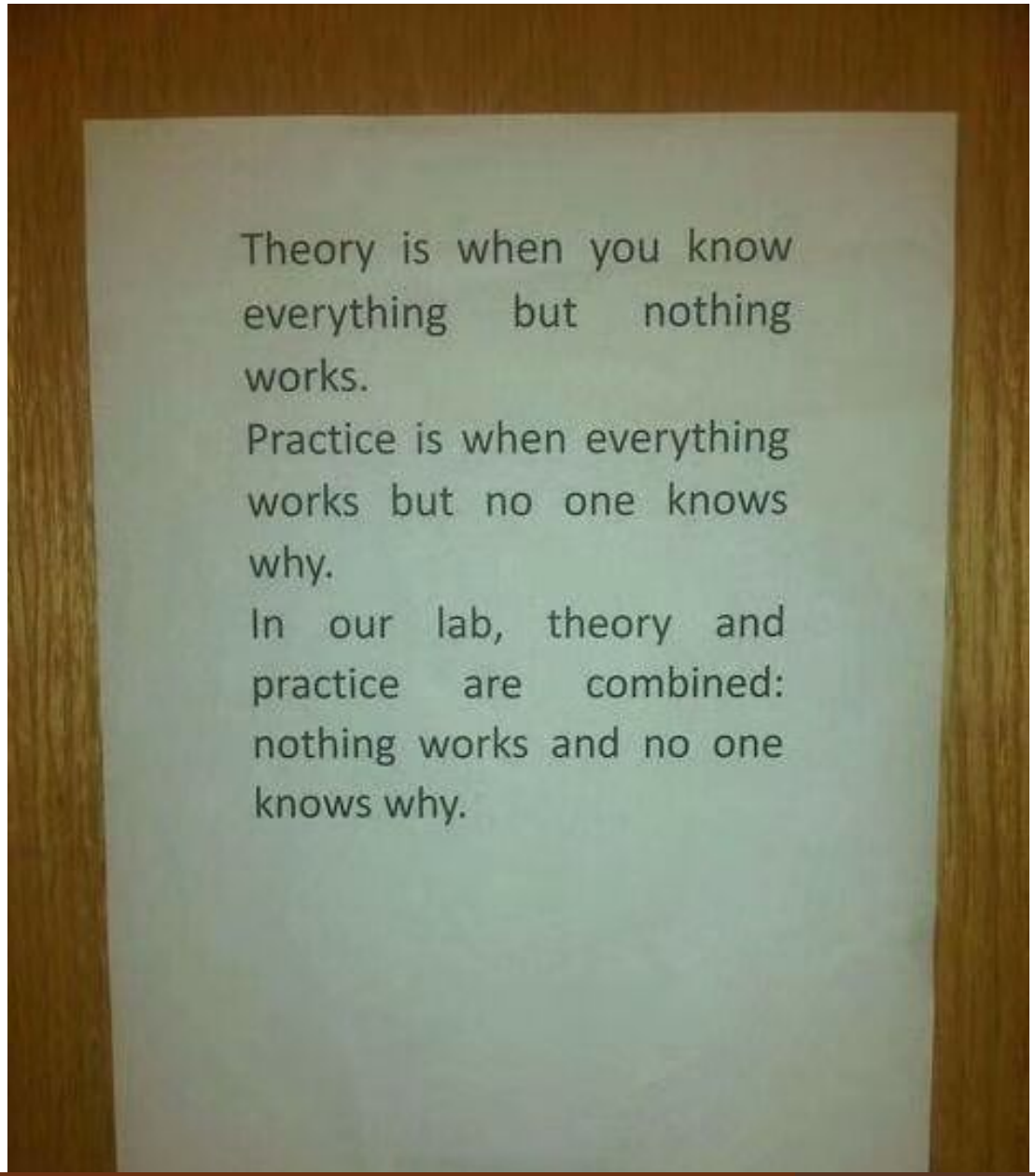
About SAT

- The first NP-Complete problem, starting NP-complete theory [Stephen Cook, 1971]
- Many theoretical results
- Many important applications

About SAT

- The first NP-Complete problem, starting NP-complete theory [Stephen Cook, 1971]
- Many theoretical results
- Many important applications
- Competitions, open source benchmarks and solvers

About SAT: theory vs. practice



About SAT: theory vs. practice

- The NP-completeness of the problem indicates that it is likely need to cost an exponential time in the worst case.

About SAT: theory vs. practice

- The NP-completeness of the problem indicates that it is likely need to cost an exponential time in the worst case.
- The only hope for a practical solver is that by being smart in the search.

About SAT: theory vs. practice

- The NP-completeness of the problem indicates that it is likely need to cost an exponential time in the worst case.
- The only hope for a practical solver is that by being smart in the search.
- Modern SAT solvers can solve formulas with up to millions of variables.

Some recent applications of SAT

- Design of Intel Core 7

Some recent applications of SAT

- Design of Intel Core 7
- Verification of driven programs in Windows 7

Some recent applications of SAT

- Design of Intel Core 7
- Verification of driven programs in Windows 7
- Math Theorem Proving
  - Proving the answer to Boolean Pythagorean Triples (PTN) problem is NO (the problem asks: can the set of natural numbers be divided into two parts, such that no part contains a Pythagorean triple (a, b, c), i.e., $a^2 + b^2 = c^2$.) [Marijn et al., 2016]

Some recent applications of SAT

- Design of Intel Core 7
- Verification of driven programs in Windows 7
- Math Theorem Proving
  - Proving the answer to Boolean Pythagorean Triples (PTN) problem is NO (the problem asks: can the set of natural numbers be divided into two parts, such that no part contains a Pythagorean triple (a, b, c), i.e., $a^2 + b^2 = c^2$.) [Marijn et al., 2016]
- Radio spectrum reallocation of USA (7-billion dollars earnings) by a software named SATFC [Leyton-Brown et al., PNAS 2017], in which a core SAT solver is our solver DCCASat.

Solving SAT

Methods for solving SAT can be classified to two classes:

- Complete methods: DPLL $\longrightarrow$ Conflict Driven Clause Learning (CDCL)
  - Guarantee the correct solution when the algorithm terminates, but it can cost unreasonable long time

Solving SAT

Methods for solving SAT can be classified to two classes:

- Complete methods: DPLL $\longrightarrow$ Conflict Driven Clause Learning (CDCL)
  - Guarantee the correct solution when the algorithm terminates, but it can cost unreasonable long time
- Incomplete methods: mainly Local search
  - Can not prove the unsatisfiability of a formula, but may find a solution fast for certain types of instances

Backtracking framework of Complete Search for SAT

- search space of a formula

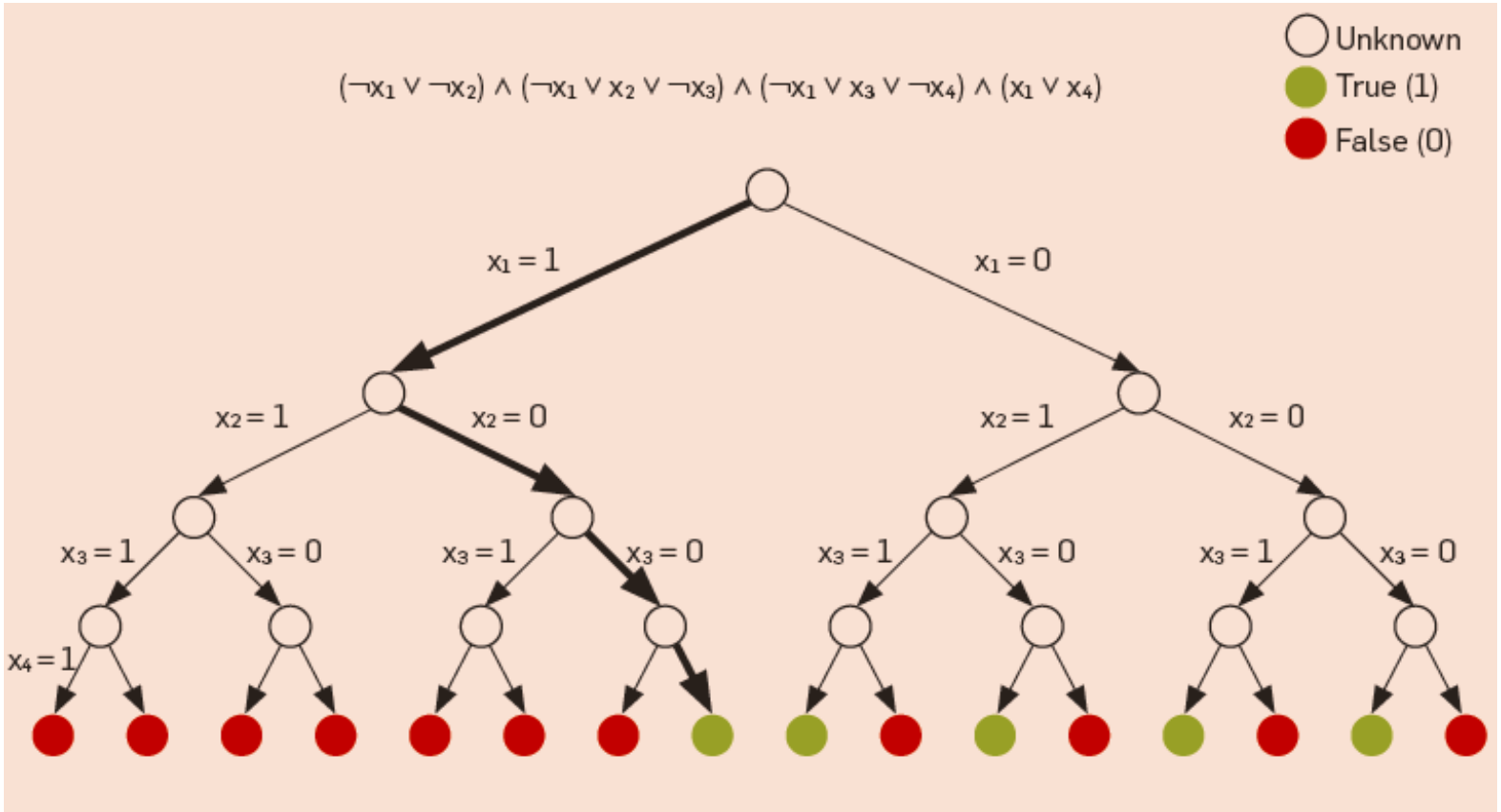


figure taken from [Sharad Malik, Lintao Zhang, CACM 2009]

Basic Rules

- Pure Literals
  - A literal is pure if only occurs as a positive literal or as a negative literal in a CNF formula.
  - The variable should be assigned to make the pure literal true.

Basic Rules

- Pure Literals
  - A literal is pure if only occurs as a positive literal or as a negative literal in a CNF formula.
  - The variable should be assigned to make the pure literal true.
  - E.g. $\varphi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_4 \vee \neg x_5)$, x_1 and $\neg x_5$ are pure literals.

Basic Rules

- Pure Literals
  - A literal is pure if only occurs as a positive literal or as a negative literal in a CNF formula.
  - The variable should be assigned to make the pure literal true.
  - E.g. $\varphi = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (x_1 \vee \neg x_3 \vee \neg x_4) \wedge (x_4 \vee \neg x_5)$, x_1 and $\neg x_5$ are pure literals.
  - A reference technique until the mid 90s; nowadays seldom used.

Basic Rules

- Unit Propagation
  - Given a unit clause, its only unassigned literal must be assigned value 1 for the clause to be satisfied.
  - E.g., for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, and $x_1 = 0$, $x_2 = 1$, then x_3 must be assigned value 0.

Basic Rules

- Unit Propagation
  - Given a unit clause, its only unassigned literal must be assigned value 1 for the clause to be satisfied.
  - E.g., for unit clause $(x_1 \vee \neg x_2 \vee \neg x_3)$, and $x_1 = 0$, $x_2 = 1$, then x_3 must be assigned value 0.
  - Very important technique even nowadays.

DPLL Algorithm

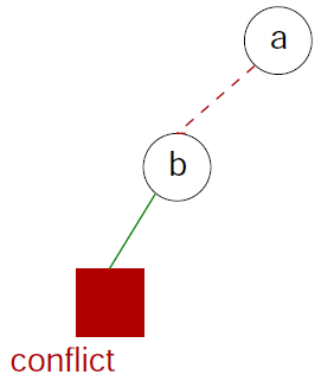
DPLL algorithm [Davis and Putnam 1960, Davis, Logemann and Loveland 1962]

- backtracking framework
- equipped with simple reasoning techniques including pure literals and unit propagations.
- heuristics for variable ordering and value ordering.

DPLL Algorithm

Example:

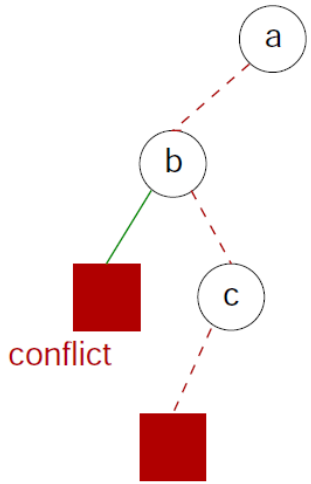
$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



DPLL Algorithm

Example:

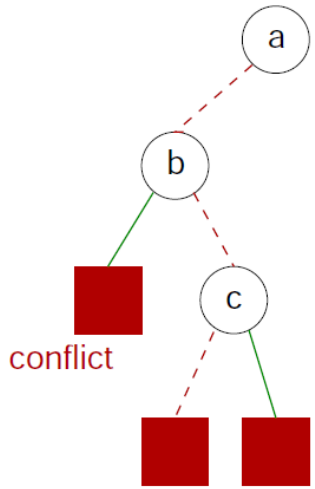
$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



DPLL Algorithm

Example:

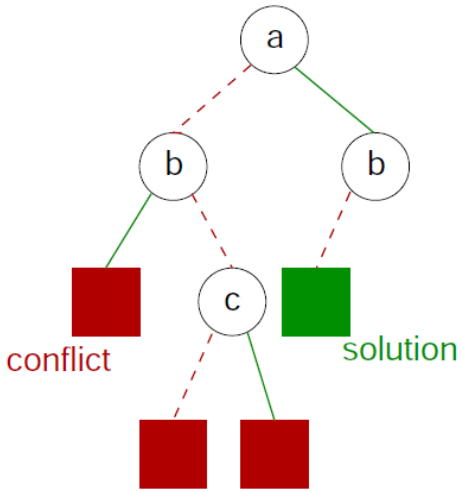
$$\begin{aligned}\varphi \quad = \quad & (a \vee \neg b \vee d) \wedge (a \vee \neg b \vee e) \wedge \\ & (\neg b \vee \neg d \vee \neg e) \wedge \\ & (a \vee b \vee c \vee d) \wedge (a \vee b \vee c \vee \neg d) \wedge \\ & (a \vee b \vee \neg c \vee e) \wedge (a \vee b \vee \neg c \vee \neg e)\end{aligned}$$



DPLL Algorithm

Example:

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



From DPLL to CDCL

Conflict Driven Clause Learning (CDCL)

- Learning and non-chronological backtracking, 1996
- Conflict Driven Clause Learning (CDCL), 1999
- Efficient data structures
- Restart

Clause Learning

Clause Learning:

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

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

  - 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

Clause Learning:

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

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

  - 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 $(a \vee c \vee f)$

Local Search for SAT

Local search for SAT: [Selman et al. 1992; Gu 1992]

Local Search for SAT

Local search for SAT: [Selman et al. 1992; Gu 1992]

- start with a complete assignment
- iteratively flip the truth value of a variable, until a satisfying assignment is found.

Local Search for SAT

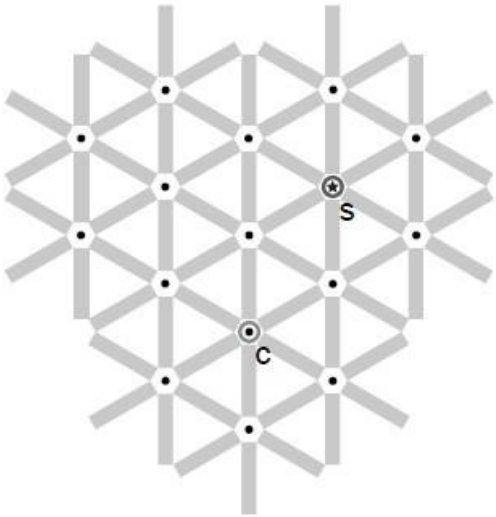
Local search for SAT: [Selman et al. 1992; Gu 1992]

- start with a complete assignment
- iteratively flip the truth value of a variable, until a satisfying assignment is found.

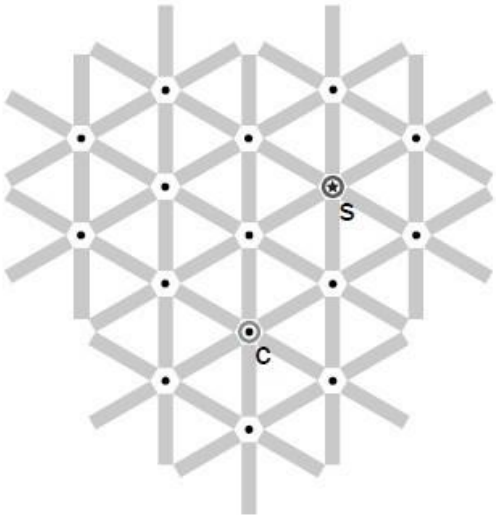
Local search is incomplete

- It cannot prove the unsatisfiability of a formula
- It cannot guarantee to find a solution even for satisfiable formula, but adding random walks can make it Probabilistic Approximate Complete.

Local Search for SAT

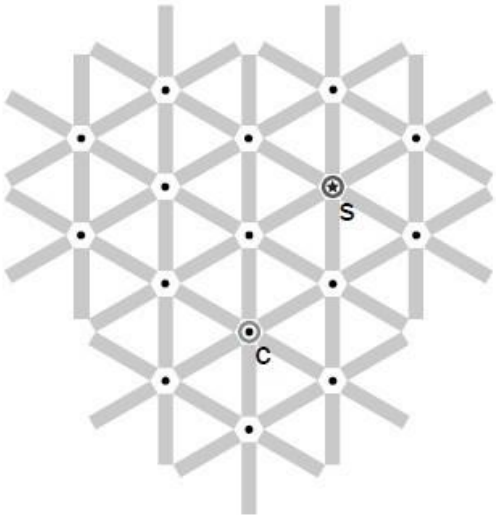


Local Search for SAT

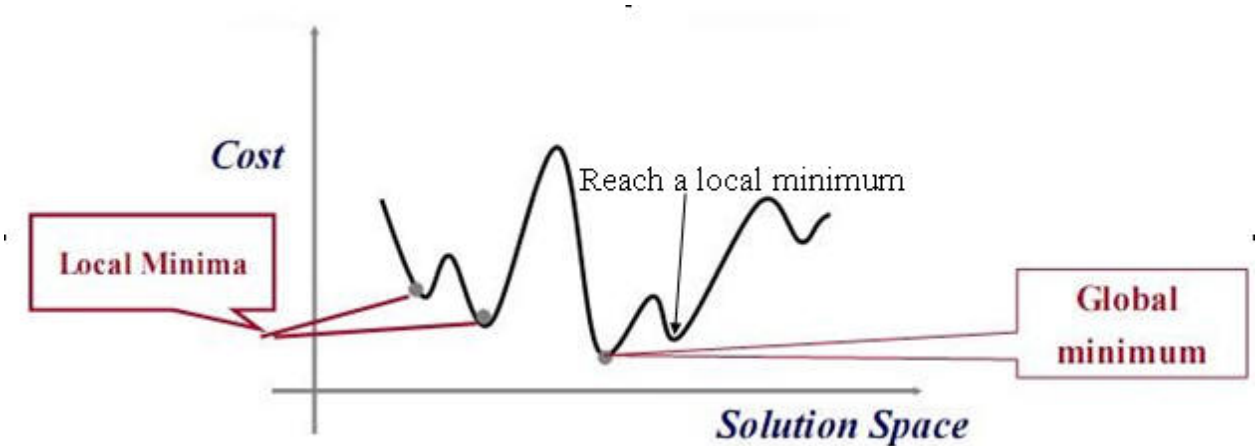


cost(α): the number of unsatisfied clauses under an assignment α.

Local Search for SAT



cost(α): the number of unsatisfied clauses under an assignment α.



Properties of Variables

Local search algorithms usually use scoring functions to pick the flipping variable.

Properties of Variables

Local search algorithms usually use scoring functions to pick the flipping variable.

- $make(x)$: the number of currently unsatisfied clauses that would become satisfied by flipping x.

Properties of Variables

Local search algorithms usually use scoring functions to pick the flipping variable.

- *make*(x): the number of currently unsatisfied clauses that would become satisfied by flipping x.
- *break*(x): the number of currently satisfied clauses that would become unsatisfied by flipping x.

Properties of Variables

Local search algorithms usually use scoring functions to pick the flipping variable.

- $make(x)$: the number of currently unsatisfied clauses that would become satisfied by flipping x.
- $break(x)$: the number of currently satisfied clauses that would become unsatisfied by flipping x.
- $score(x)$: $cost(F, \alpha) - cost(F, \alpha')$, i.e., the number of currently unsatisfied clauses minus the number of unsatisfied clauses if x were to be flipped.

Properties of Variables

Local search algorithms usually use scoring functions to pick the flipping variable.

- $make(x)$: the number of currently unsatisfied clauses that would become satisfied by flipping x.
- $break(x)$: the number of currently satisfied clauses that would become unsatisfied by flipping x.
- $score(x)$: $cost(F, \alpha) - cost(F, \alpha')$, i.e., the number of currently unsatisfied clauses minus the number of unsatisfied clauses if x were to be flipped.
- It is easy to see that $score(x) = make(x) - break(x)$.

Scoring Functions of Variables

A scoring function can be a simple variable property or any mathematical expression with one or more properties.

- $break(x)$
- $score(x)$
- $age(x)$
- $score(x) + age(x)/T$
- $score(x)^a$
- $b^{score(x)}$
- ...

The Cycling Problem of Local Search

Cycling problem, i.e., revisiting candidate solutions

The Cycling Problem of Local Search

Cycling problem, i.e., revisiting candidate solutions

- Cycling is an inherent problem of local search: local search does not allow to memorize all previously visited parts of the search space.

The Cycling Problem of Local Search

Cycling problem, i.e., revisiting candidate solutions

- Cycling is an inherent problem of local search: local search does not allow to memorize all previously visited parts of the search space.

A key factor degrading the performance of local search

- wastes time on revisiting
- prevents it from getting out of local minima

Previous Methods

- Naive methods
  - Random walk
  - allow non-improving steps with a probability (simulated annealing)
  - Restart

Previous Methods

- Naive methods
  - Random walk
  - allow non-improving steps with a probability (simulated annealing)
  - Restart
- The tabu mechanism forbids reversing the recent changes, where the strength of forbidding is controlled by a parameter called tabu tenure [Fred Glover, 1989].

Configuration Checking

Address cycling problem by Configuration Checking (CC) [Artif. Intel. 2011].

CC is found effective for the following types of problems:

- Assignment Problems: to find an assignment to all variables such that satisfies the constraints (and optimized).
- Subset Problems: to find a subset from a universe set such that satisfies the constraints (and optimized).

Configuration Checking for SAT

CC says: for a variable, if after it was flipped, its circumstance has not changed, then it should not change its value back.

Configuration Checking for SAT

CC says: for a variable, if after it was flipped, its circumstance has not changed, then it should not change its value back.
first time to consider variables' circumstance!

Configuration Checking for SAT

CC says: for a variable, if after it was flipped, its circumstance has not changed, then it should not change its value back.
first time to consider variables' circumstance!

CC relies on the concept of the "configuration" of variables, which is some form of circumstance of the variables.

Definition

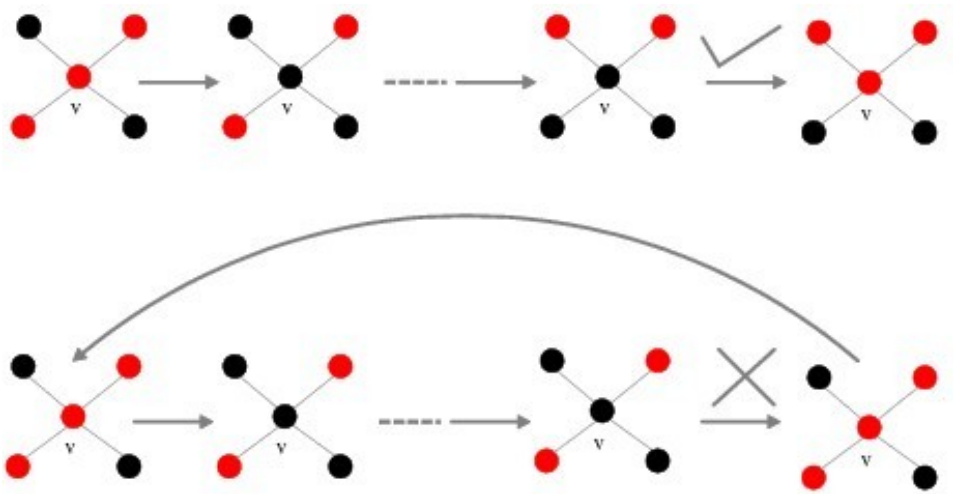
The *configuration* of a variable x is a vector C_x consisting of truth value of all x's neighboring variables.

Intuition of CC

Configuration Checking for SAT: For a variable x, if the configuration of x has not changed since x's last flip, then x is forbidden to be flipped.

Intuition of CC

Configuration Checking for SAT: For a variable x, if the configuration of x has not changed since x's last flip, then x is forbidden to be flipped.



reduce local structure cycles

Variants of CC for SAT

A typical CC strategy for SAT is Neighboring Variables based CC.

We can have variants of CC strategy by defining configuration in different ways.

Variants of CC for SAT

A typical CC strategy for SAT is Neighboring Variables based CC.

We can have variants of CC strategy by defining configuration in different ways.

Definition

In Clause States based CC (CSCC), the configuration of a variable x is a vector that consists of the states of all the clauses in which x appears.

Combination of CC strategies

We can also combine different CC strategies in one algorithm.

Double Configuration Checking (DCC) for SAT:

Lemma

If a variable is configuration changed w.r.t. CSCC criterion, then it is is configuration changed w.r.t. NVCC, while the reverse is not true.

DCC heuristic: first selects a CSCC variable; if no such variable, selects a NVCC variable.

Success of CC for SAT

- Many participant solvers use CC in SAT Competitions from 2012, including 10 medal-awarded ones.
- AAAI 2013 Tutorial Forum: "It is outstanding. It is likely changing the game."
- Used in solving real world application projects.

CC and variable neighborhood

The effectiveness of CC is related to the neighborhood of variables.

Theorem

For a random k-SAT formula $F_k(n, m)$, if $ln(n-1) < \frac{k(k-1)r}{(n-1)}$, then CC degrades forbids only one variable each step.

When CC Becomes Ineffective

$f(n) = ln(n-1) - \frac{k(k-1)r}{n-1}$ is a monotonic increasing with n $(n > 1)$.
$f(n) < 0$ iff $n \leq \lfloor n^* \rfloor$, where n^* is a real number s.t. $f(n^*) = 0$.

When CC Becomes Ineffective

$f(n) = ln(n-1) - \frac{k(k-1)r}{n-1}$ is a monotonic increasing with n $(n > 1)$.
$f(n) < 0$ iff $n \leq \lfloor n^* \rfloor$, where n^* is a real number s.t. $f(n^*) = 0$.
For phase-transition area of k-SAT, such n^* values are as follows.

Formulas	3-SAT ($r = 4.2$)	4-SAT ($r = 9.0$)	5-SAT ($r = 20$)	6-SAT ($r = 40$)	7-SAT ($r = 85$)
n^*	11.652	32.348	90.093	223.095	564.595

Table: The n^* value such that when $n < n^*$, the CC strategy degrades to a trivial case.

Other works

Some of my other works on SAT solving:

- subscore: a scoring function considering the satisfaction degree of clauses.

Other works

Some of my other works on SAT solving:

- subscore: a scoring function considering the satisfaction degree of clauses.
- Combining local search with reasoning, e.g., unit propagation, clause learning.

Other works

Some of my other works on SAT solving:

- subscore: a scoring function considering the satisfaction degree of clauses.
- Combining local search with reasoning, e.g., unit propagation, clause learning.
- Automatic algorithm configuration.

Summary

Messages to take:

- Gap between theory and practice of SAT

Summary

Messages to take:

- Gap between theory and practice of SAT
- Current state of SAT solving

Summary

Messages to take:

- Gap between theory and practice of SAT
- Current state of SAT solving
- What is CDCL

Summary

Messages to take:

- Gap between theory and practice of SAT
- Current state of SAT solving
- What is CDCL
- Local search

Summary

Messages to take:

- Gap between theory and practice of SAT
- Current state of SAT solving
- What is CDCL
- Local search
- Configuration checking

Thank you!