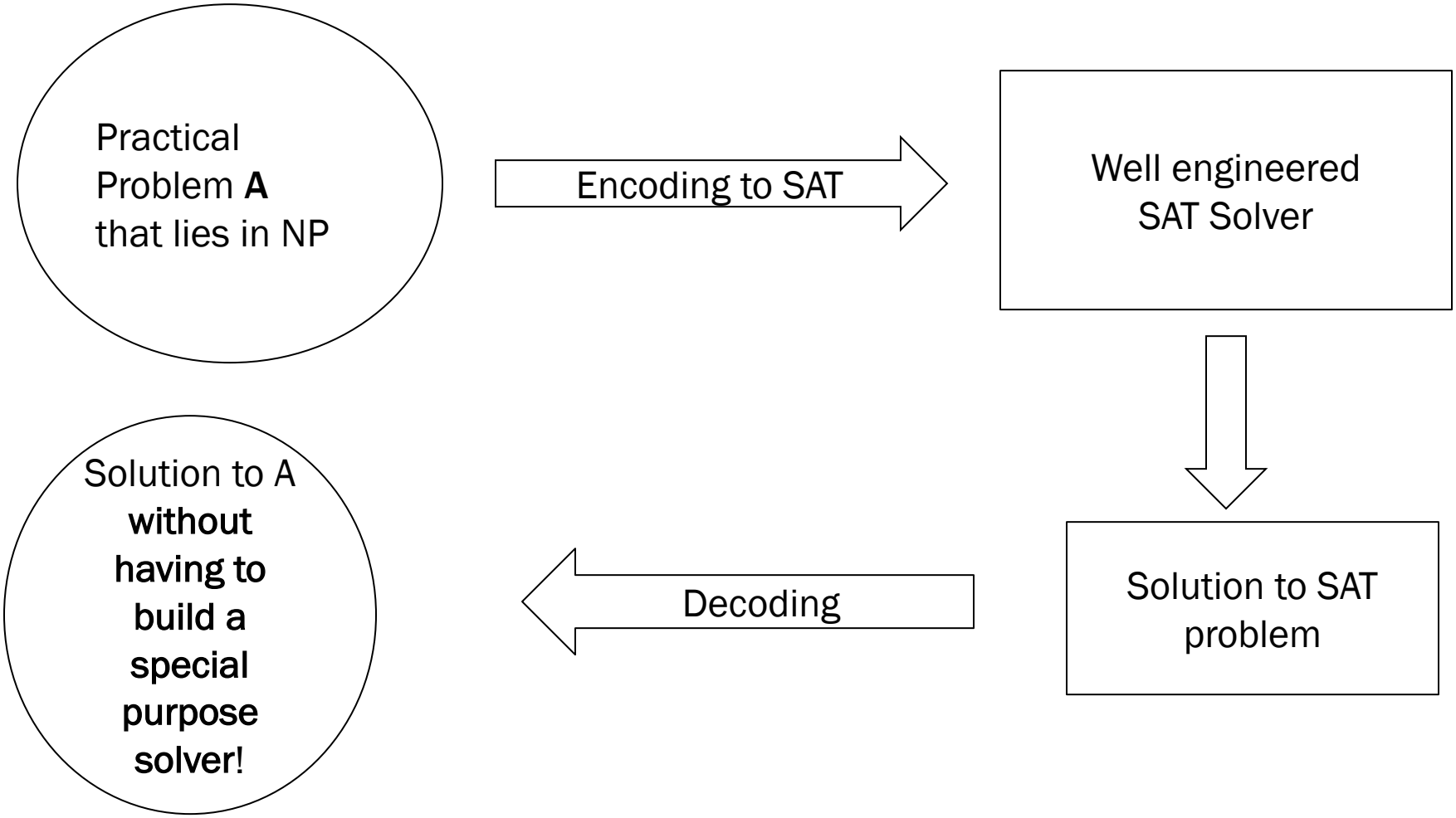
Modern Sat Solving
CDCL Tutorial
Singapore Winter School 2019

Fahiem Bacchus, University of Toronto

Why Sat solving?



Why Sat Solving?

- Can this approach be successful?
- Yes, we can solve many practical problems with this approach.
- Evidence with modern SAT solvers indicate that in fact this approach can sometimes offer significant **performance improvements** over developing problem specific software.

- In fact this approach can be successful for other complexity classes
  - Later we will discuss solvers for MaxSat which is problem complete for the class FP^{NP} (the set of problems that be solved in polynomial time given access to an NP oracle).

Conjunctive Normal Form

Modern SAT solvers work with propositional formulas expressed in Conjunctive Normal Form (CNF)

CNF: a conjunction of clauses, each of which is a disjunction of literals, each of which is either a propositional variable or the negation of a propositional variable.

$$(p_1 \vee \neg p_2 \vee p_3) \wedge (p_2 \vee \neg p_5) \wedge (p_2 \vee \neg p_6) \wedge (p_4 \vee p_5) \wedge (\neg p_3)$$

We typically write this in abbreviated form:

$$(p_1 , \neg p_2, p_3)(p_2, \neg p_5)(p_2, \neg p_6)(p_4, p_5)(\neg p_3)$$

Semantics

Truth Assignments (Models)

1. Truth assignment π: map each propositional variable to True/False (0,1)

$$p_i \rightarrow \{0, 1\}$$

2. $\pi(\neg p) = 1$ if $\pi(p) = 0$
   $= 0$ if $\pi(p) = 1$
3. $\pi(c) = 1$ if $\pi(l) = 1$ for at least one literals $l \in c$
   $= 0$ otherwise

$\pi(F) = 1$ if $\pi(c) = 1$ for all clauses $c \in F$

Satisfiability

CNF Formula F

F is satisfiable if there exists a truth assignment π such that $\pi(F) = 1$. Unsatisfiable otherwise.

Written as $\pi \models F$

f is a logical consequence of F if for all truth assignment π such that $\pi \models F$ **we have that** $\pi \models f$

Written as $F \models f$

Obvious simplifications

1. A clause with clashing literals in it is true under any truth assignment. Such clauses are called **tautological.** Such clauses can be removed from the CNF
2. Duplicate literals are irrelevant and can be removed
3. We say that a clause c is **subsumed by** another clause c' if c' is a subset of c
   - Any truth assignment that satisfies c must also satisfy c'
   - Subsumed clauses can be removed from the CNF

Some Observations

Notes:

1. Each clause serves to eliminate some set of truth assignments (i.e., these truth assignments cannot be models of the CNF.
   - E.g., (a, b, $\neg$c) eliminates all truth assignments π such that $\pi(a) = 0$, $\pi(b) = 0$, and $\pi(c) = 1$
   - Shorter clauses eliminate more truth assignments
2. By convention no truth assignment satisfies the empty clause '()'
3. Satisfiability is NP complete

CSC2512: CNF

CNF

a	b	c	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	
1	1	0	0
1	1	1	

(a)

(-a, c)

Determining if there is a one anywhere (satisfiable) for **F** becomes combinatorial as each clause makes a different set of truth assignments unsatisfying.

Converting to CNF

Any propositional formula can be converted to CNF with no more than a polynomial increase in size **by introducing new variables** (Tseitin 1970)

If we don't introduce new variables—we can have an exponential increase in the size of the formula.

Encodings

Encodings: CNF is not a natural language for most applications. Various domains have different “standard” languages.

- Automated Planning: STRIPS or ADL actions specified with first-order variables
- Hardware: Circuits
- Software: Various specification languages (logics with extensions).

Specialized techniques have also been developed to encode problems expressed in these languages in CNF. The encoding used can have a tremendous impact on how easy it is to solve the CNF.

Resolution

Reasoning with CNF

CNF is used in modern SAT solvers mainly because there is a very simple reasoning rule that can be efficiently implemented.

Definition: **Resolution**
Two clauses with a single clashing literal can be resolved:

$$R[(A, x), (B, -x)] = (A,B)$$

(where A and B are sets of literals). We assume that duplicate literals are removed.

- If A and B have more than one clashing pair of literals the result will be a tautology

Resolution

Resolution is **sound**: Any truth assignment that satisfies c and c' must satisfy R[c,c'].

That is resolution generates **logical consequences.**

$$\textbf{if } \pi \models \mathbf{c} \wedge \mathbf{c'} \textbf{ then } \pi \models R[c,c']$$

Resolution Refutations

A Resolution Proof of a clause c_n from a CNF F

A sequence of clauses $c_1, c_2, \ldots, c_n$ such that:

1. Each c_i is either
   1. A member of the set of clauses F
   2. or was derived by a resolution step from two prior clauses in the sequence c_j and c_k ($j, k < i$)

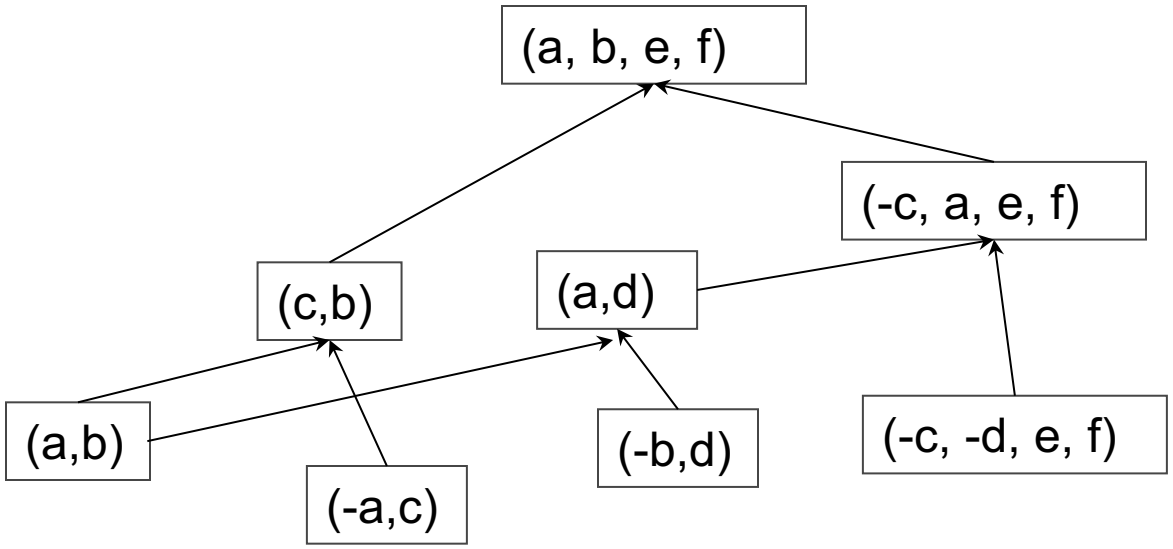
The sequence can also be represented as a DAG (directed acyclic graph).

Resolution Proofs

C = {(a, b) (-a, c) (-b, d) (-c, -d, e, f)}

There is a resolution proof of (a, b, e, f) from C:

(a, b), (-a, c), (c, b), (-b, d), (a, d), (-c, -d, e, f) (-c, a, e, f)
(a, b, e, f)



Resolution Refutations

Definition: **Resolution Refutation** of a CNF F is a resolution proof of the empty clause '()' from F.

From soundness, any truth assignment satisfying F must satisfy the empty clause, but no truth assignment satisfies the empty clause ➔ A refutation proves that F is unsatisfiable

Resolution Refutations

Resolution is **Refutation Complete:**

If F is unsatisfiable there exists a resolution refutation of C.

Equivalently if $\mathbf{F} \models \mathbf{f}$ then we can derive the empty clause from $\mathbf{F} \wedge \neg\mathbf{f}$ using resolution.

DLL (DPLL)

Davis, Logemann and Loveland [1962] introduced a procedure for solving SAT based on backtracking. (Became commonly known as DPLL)

Modern Conflict Directed Clause Learning (CDCL) algorithms can be viewed as generalizations of DPLL

DPLL

DPLL(π, F) // Initially F is the input formula.
π is an empty set of literals (truth assignment)

If **F** is empty
  return (SAT,π) (π is a satisfying assignment)

If **F** contains an empty clause
  return (UNSAT,$\varnothing$)

else choose a variable **v** in **F** //Prefer a **v** appearing
  //in a unit clause if one exists

  F' = F|$_v$ //Reduce F

  (SAT?,π') = **DPLL(π + v, F')**

    if SAT? == SAT return (SAT,π')

  F' = F|$_{-v}$

  return DPLL(π + -v, F')

DPLL

Reduction:

$\mathbf{F}|_l$ F reduced by literal l

Remove all clauses of F that contain l
(they are satisfied)

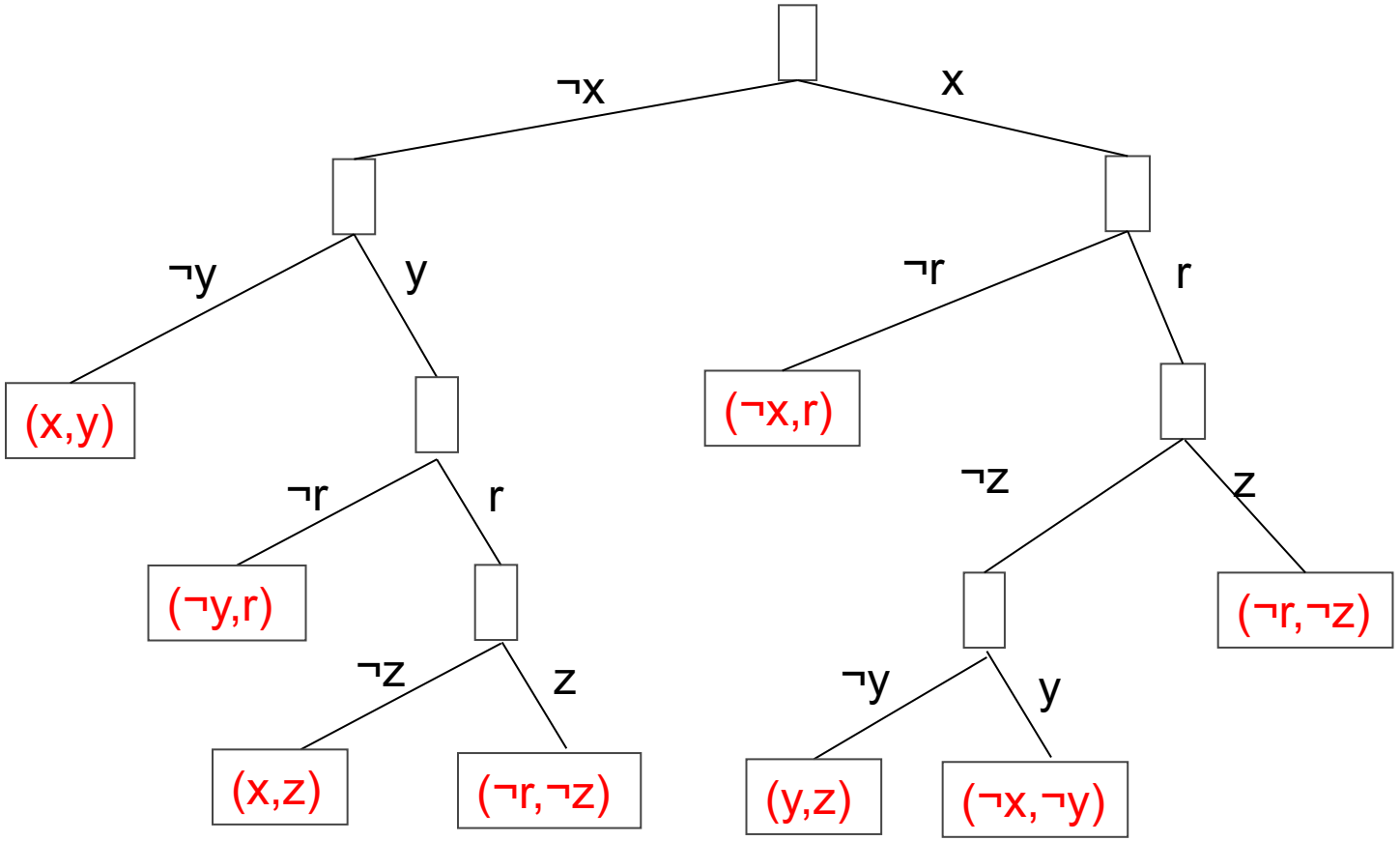
Remove –l from all remaining clauses
(-l can no longer satisfy them)

$\{(a, b, -d), (d, c, e), (g, h, e)\}|_{-d}$
$= \{(c, e), (g, h, e)\}$

DPLL

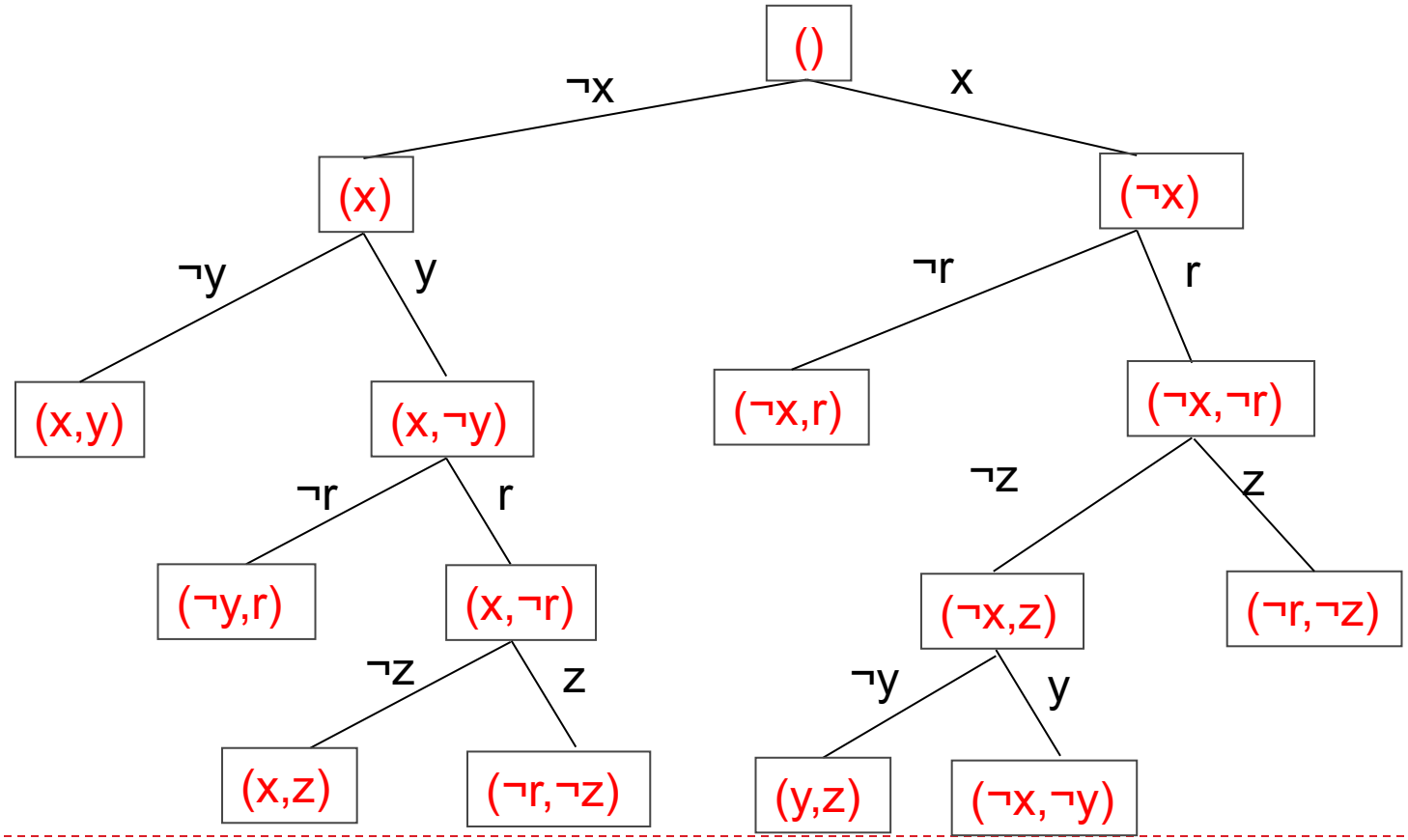
Example:

F = (¬x, r), (¬y, r), (x, z), (y, z), (x, y), (¬x, ¬y), (¬z, ¬r)



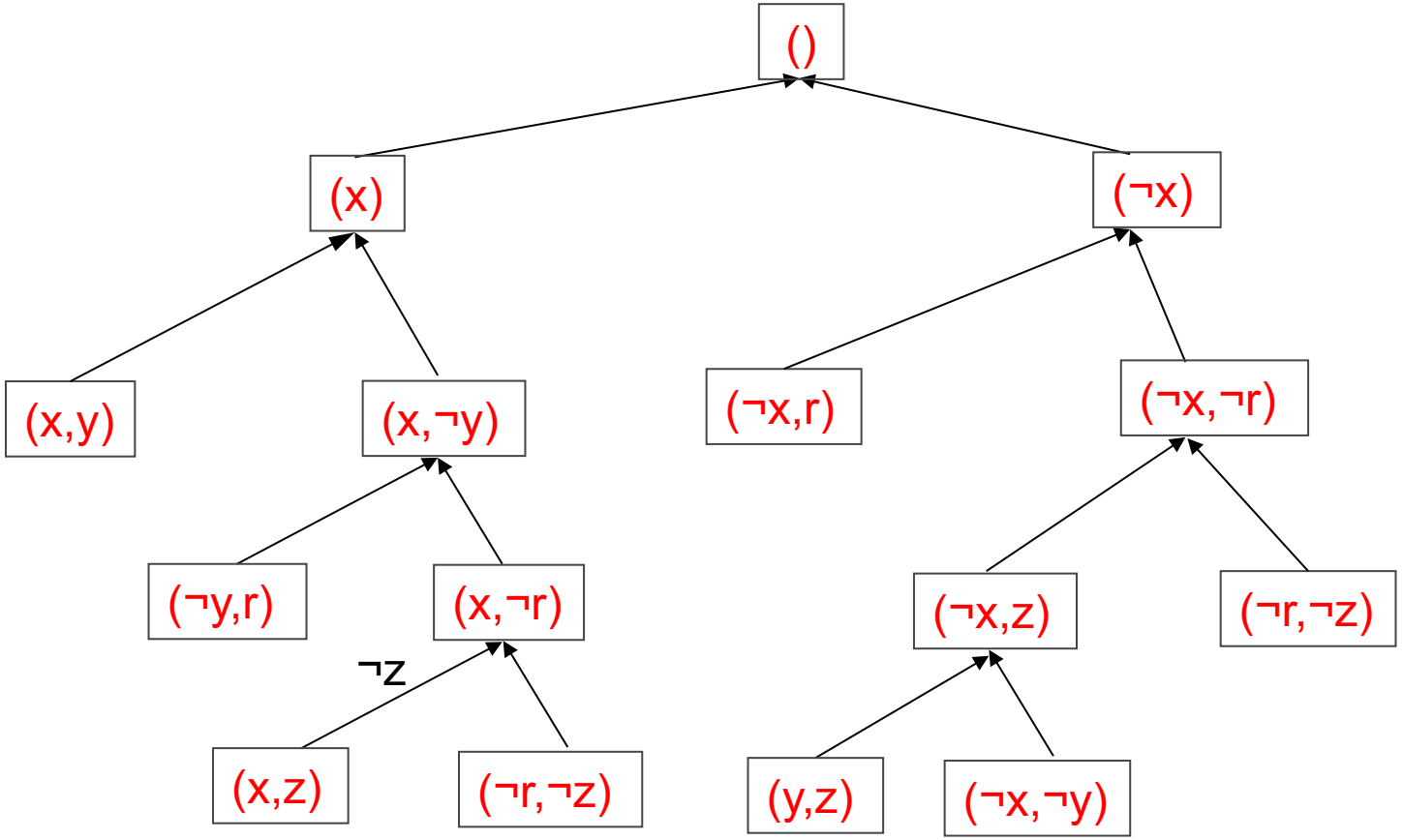
DPLL

From every execution of DPLL yielding UNSAT we can extract a resolution refutation. Label each node with resolvent of its two children



DPLL

The resultant resolution DAG is a tree.



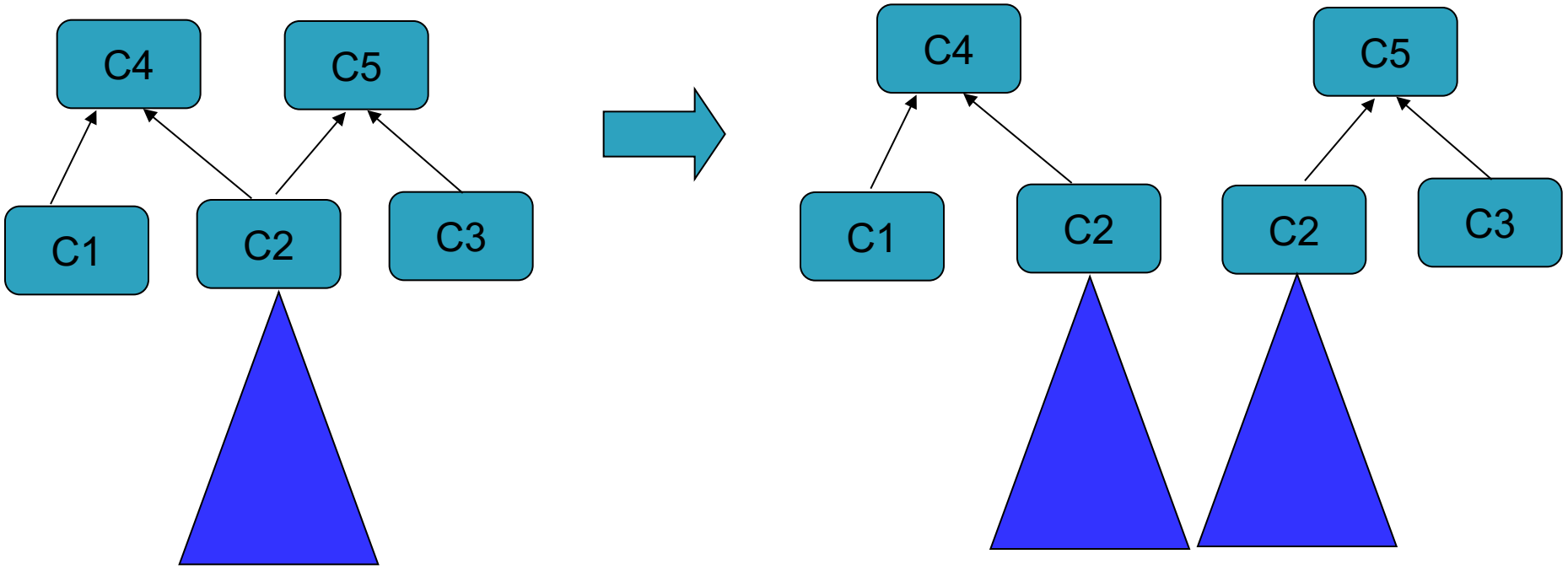
Tree Resolutions

Tree Resolutions

A special form of resolution proof in which the resolution DAG of each refutation is a tree (we are allowed to use the input clauses more than once).

Tree resolution is sound (it is uses the resolution rule) and **refutation complete**. Any resolution refutation can be converted to a tree resolution.

Conversion to Tree Resolution



Tree Resolutions

Tree Resolutions have no memory, other than input clauses if clause has to be used in more than one resolution step it has to be rederived for each use.

As a result DPLL will often be very inefficient.

P-Simulation

Proof Systems (Cook & Reckhow)

- A **proof system** for a language **L** is a polynomial time algorithm **PC** s.t.
  - For all inputs **F**
    F $\in$ **L iff** there exists a string **P** s.t. **PC accepts input (F,P)**

EXAMPLE

- **L** is the set of **unsatisfiable** CNF formulas. **F** is a sample CNF, and we want to test if **F** is unsatifiable.
- **P** is a proof that **F** is UNSAT, this proof is valid if there is a proof-checking algorithm (verifier) **PC** that runs in time polynomial in the size of **P** and **F**
- The string **P** is a proof, e.g., a resolution refutation. But other proof systems exist that verify other type of proofs.

P-Simulation

Proof Systems.

- The **complexity** of a **proof system, PC** for a language **L** is a function

$$f(n) = \max_{F \in L, |F| = n} \min_{P : s.t. PC\ accepts\ (F, P)} |P|$$

- The smallest proof of any F that is accepted by the proof system. f(n) is how the maximum smallest proof grows as the length of F grows.

P-Simulation

Proof Systems.

- Given two proof systems PC_1 and PC_2 we say that PC_1 **p-simulates** PC_2 if there is a polynomially computable function f such that for any proof P_2 of PC_2 (i.e., proof accepted by PC_2) $f(P_2)$ is a proof of PC_1.

- In other words any proof of PC_2 can be converted to a proof of PC_1 with at most a polynomial increase in size

Tree Resolution

- **Tree** resolution cannot p-simulate general resolution. That is, there exists formulas **F** that have poly-sized resolution proofs but whose whose shortest tree/ordered resolution proofs are exponential in size.

- Since DPLL's search tree corresponds to a tree resolution this means that DPLL must run in exponential time on such **F, even though F is "easy" for general resolution.**

CDCL Solvers

Modern SAT solvers

1. Based on DPLL
2. More efficient implementation methods.
3. Clause learning which gets around the memory less limitation of tree resolution.
    1. This is done by explicitly keeping track of the clauses falsified at the leaves and the clauses associated with the nodes arising from resolution steps.
4. Uses the learnt clauses to heuristically guide the solver's search (Conflict Directed)
5. Other important advances

Detecting Unit and Empty Clauses Efficiently

DPLL(π, F)

If **F** is empty
  return (SAT,π) (π is a satisfying assignment)

If **F** contains an empty clause //F restricted by prior //assignments
  return UNSAT

else choose a variable **v** in **F preferring v in unit clauses**
//Need to find unit clauses

F' = F|$_v$
  (SAT?,π') = **DPLL(π + v, F')**
    if SAT? == SAT return (SAT,π')
  F' = F|$_{-v}$
  return DPLL(π + -v, F')

Detecting Units Efficiently

We need fast ways to find units and empty clauses in $F|_v$

Computing $F|_v$ and then restoring F on backtrack would be too time consuming (F can have millions of clauses)

Detecting Units Efficiently

Clauses are not removed and literals are not removed from clauses. Rather literals are made true/false.

1. A clause is considered to be removed if one of its literals is true. The clause is satisfied.

   (x, ¬y, z)

2. A clause is empty if all of its literals are false.

   (x, ¬y, z)

3. A clause is unit if it is not satisfied and all but one of its literals are false.

   (x, ¬y, z)

We want to detect these cases efficiently.

Unit Propagation

Once a clause is detected to be unit

(x, ¬y, z)

The SAT solver must set the remaining literal to True.

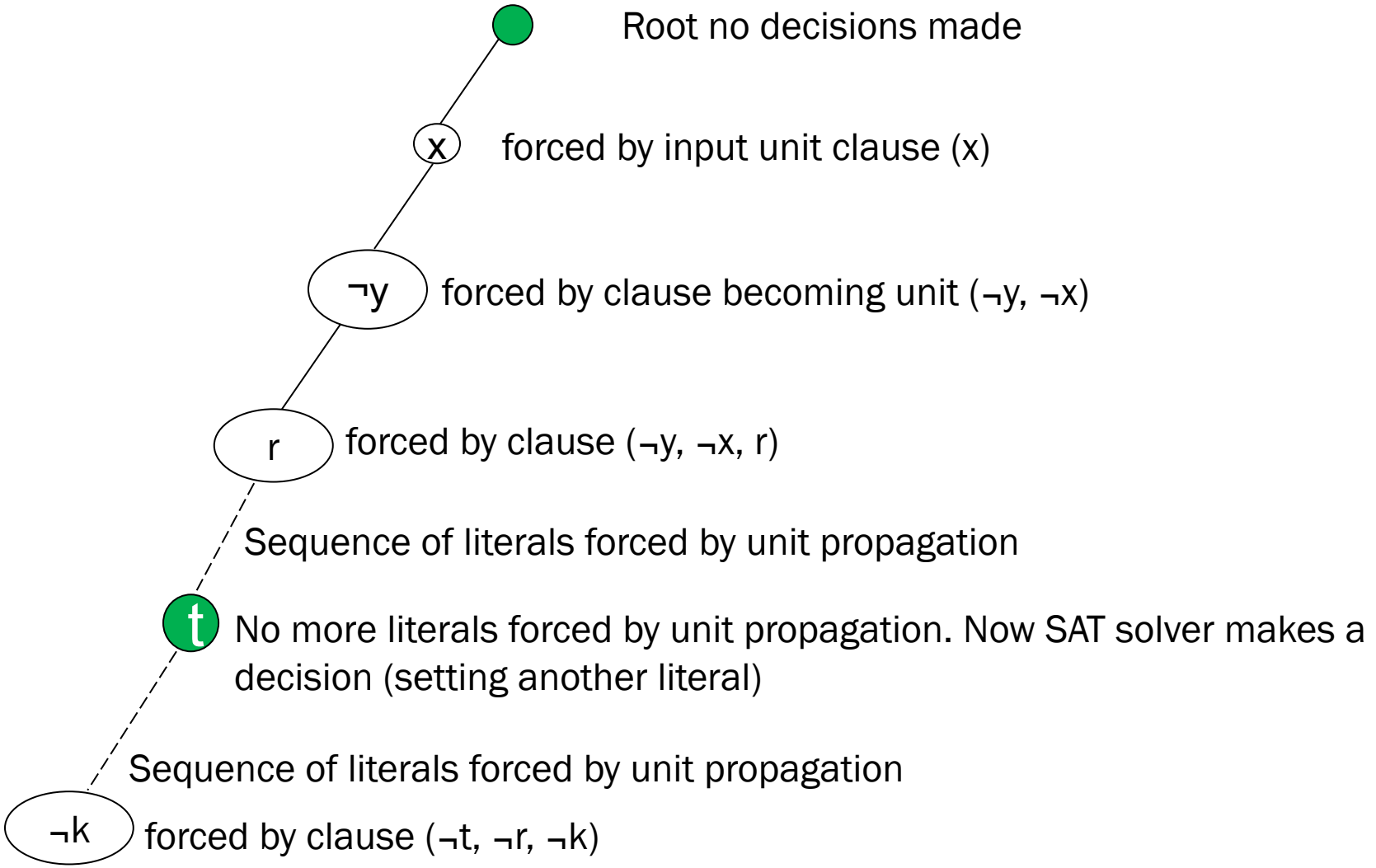
x

On this literal is set to True some other clause might become unit

(¬x, ¬y, r)

This process run to completion (setting all literals forced by unit clauses) is called Unit Propagation

Search (Trail)



Detecting Units Old way

For each literal keep a list of clauses it appears in.

Keep a count of the false literals in the clause.

If x is made false, increment the count for every clause it is in. If that count is equal to the clause length -1 the clause has become unit.

Examine the clause to find the literal it implies

Requires work for every clause x appears in

Requires work to restore the counts on backtrack.

Detecting Units New way

Two clauses are selected from each clause to be **watch literals.**

Each literal has a list of clauses it watches.

So whenever a literal becomes false we check **only** the clauses it watches (a fraction of the clauses it appears in).

Make x false:

Examine the clauses that x watches:

- If the other watch is True, do nothing (clause is satisfied)
- Else find a non-true literal y in the clause that is not the other watch.
  - If there is no such y,
    if other watch is unset the clause is unit
    if the other watch is False the clause is empty
  - Else (found y)
    Remove clause from x's watch list, add it to y's watch list (make y a new watch.

Watch Literals

So to update with a newly false literal we need only check about a fraction of the clauses the literal appears in (those it watches).

No work needs to be done on backtrack—if the watches are valid, they will remain valid on backtrack.

Clause Learning

When we find an empty clause (falsified clause) DPLL will backtrack—we have hit a deadend.

CDCL also backtracks but first learns and remembers a new clause. This new clause will block this deadend and hopefully other deadends.

Clause Learning

- X
  - A
  - ¬B
  - C
- ¬Y
  - D
  - ¬E
  - F
- Z
  - H
  - I
  - ¬J
  - ¬K

(K,¬I,¬H, ¬F,E, ¬D,B)

- X,Y,Z: Decision Variables.
- A,¬B,C,D,¬E,F,H,I,¬J,¬K: forced by unit propagation
- (K,¬I,¬H, ¬F,E, ¬D,B): **Conflict Clause**

Clause Learning

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- Each forced literal was forced by some clause becoming unit.

Clause Learning

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

Each clause reason contains

1. One true literal on the path (the literal it forced)
2. Literals falsified higher up on the path.

Clause Learning

- X
  - A ← …
  - ¬B ← (¬B, ¬A)
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- We can resolve away any sequence of forced literals in the conflict clause.
- Such resolutions always yield a new falsified clause.

1. (K,¬I,¬H,¬F,E, ¬D,B), (D,B,Y) ➔ (K,¬I,¬H,¬F,E,B,Y), (¬B, A) → (K,¬I,¬H,¬F,E,A,Y)
2. (K,¬I,¬H,¬F,E, ¬D,B), (¬K,¬I,¬H,E,B) ➔ (¬I,¬H,¬F,E, ¬D,B)
3. (K,¬I,¬H,¬F,E, ¬D,B), (H,B,E,¬Z) ➔ (K,¬I,¬F,E,¬D,B,¬Z)
4. …

Clause Learning

- Any forced literal x in any conflict clause can be resolved with the reason clause for –x to generate a new conflict clause.
- If we continued this process until all forced literals are resolved away we would end up with a clause containing decision literals only (**All-decision clause**).
- But empirically the all-decision clause tends not be very effective.
  - Too specific to this particular part of the search to be useful later on.

1-UIP clauses

- The standard clause learned is a 1-UIP clause
- This continually involves resolves the trail deepest literal in the conflict clause until there is only one literal left in the clause that is at the deepest level.
  - Since every resolution step replaces a literal by literals falsified higher up the trail, we must eventually achieve this condition
  - The sole remaining literal at the **deepest level** is called the **asserted literal.**

1-UIP Clause

- X
  - A ← …
  - ¬B ← (¬B, ¬A)
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

1. (K,¬I,¬H, ¬F,E, ¬D,B), (¬K,¬I,¬H,E,B)
   ➔ (¬I,¬H, ¬F,E, ¬D,B)
2. (¬I,¬H, ¬F,E, ¬D,B), (I,¬H,¬D,¬X)
   ➔ (¬H, ¬F,E, ¬D,B,¬X)

The 1-UIP clause shows that ¬H was actually implied at the previous decision level.

But before the SAT solver didn't know this.

1-UIP clauses

- A **1-UIP** clause is sometimes called an **empowering clause**. Once we have it, UP will force a literal that it wasn't able to before.

1-UIP clauses

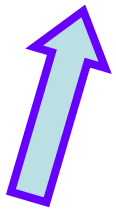
- The 1-UIP clause forces its asserted literal at a prior decision level (if we had the clause before we would have forced the asserted literal before).
- We backtrack so as to fix the trail to account for the new 1-UIP clause.
- The asserted literal is forced as soon as all other literals in the clause became false. The **assertionLevel** is the **second deepest** decision level in the clause (the asserted literal is at the deepest level)
- So we backtrack to that level (not undoing the decision or anything forced at that level), add the asserted literal to the trail, with the 1-UIP clause as its reason, then apply UP again.

1-UIP Clause

- X
  - A ← …
  - ¬B ← (¬B, ¬A)
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
- Z
  - H ← (H,B,E,¬Z)
  - I ← (I,¬H,¬D,¬X)
  - ¬J ← (¬J,¬H,B)
  - ¬K ← (¬K,¬I,¬H,E,B)

(K,¬I,¬H, ¬F,E, ¬D,B)

- X
  - A ← …
  - ¬B ← …
  - C ← …
- ¬Y
  - D ← (D,B,Y)
  - ¬E ← …
  - F ← …
  - ¬H ← (¬H,¬F,E, ¬D,B,¬X)



More unit propagation

(¬H, ¬F,E, ¬D,B,¬X)

1-UIP clauses

- Note that we are jumping back across incompletely tested decisions.
  - We backtracked over **Z**, but we don't know if **¬Z** might not have lead to a solution.
  - All we know is that the trail is now patched to account for the newly learnt clause
  - Search is no longer "systematic" like DPLL
  - Instead completeness comes from learning clauses.
- (a) it is cheap to backtrack, (b) going back far enough to fix the trail makes the implementation more efficient, (c) allows the search to explore a different area of the space.

1-UIP clauses

- If the 1-UIP clause is unit we go back to level zero—before any decision. So clause learning can generate a number of restarts.

VSIDS Heuristic

- Heuristic for selecting next decision literal (variable)
- Variable State Independent Decaying Sum
- Intuitions vary: but VSIDS is thought to encourage resolutions involving most recently learnt clauses.
  - A counter for each variable. Increment the counter of all variables in **each** clause that is used in the 1-UIP clause learning process.
  - Periodically divide all counts by 2.
  - Pick the unassigned variable with highest count at each decision
- Low overhead (counters updated only for variables in conflict). Variables kept on heap ordered by counter.
- Causes the SAT solver to branch on variables that appeared in recent learnt clauses.

Phase Saving/Restarts

Phase saving

- We decide to branch on a variable: what literal to try first?
- Use the literal that was the most recent setting of the variable on the trail.

Restarts

- Periodically restarting the solver (undoing all decisions) is useful.
  - Various strategies have been investigated for when to restart.
  - Note that because of phase saving and the fact that the VSIDS scores are unchanged, restarts tend to put back the same literals on the trail---but in a different order.

Resolution Power

- With these various features it can be show that CDCL solvers (Conflict Driven Clause Learning) are no longer limited to **tree-resolution** instead they can **p-simulate general resolution**
- Remains an open question whether or not CDCL **without restarts** is as powerful as general resolution.

Other Techniques

1. **Clause reduction:** Once we have the 1-UIP clause we can try to resolve away further literals in such a way that the clause is reduced in size.
2. **Forgetting Learnt Clauses:** We remember how many different decision levels appear in the learnt clause. This is called the LBD number for the learnt clause.
   a) Every 10,000 learnt clauses we sort all of the learnt clauses by LBD, and remove that ½ that has highest LBD (but keep all clauses with LBD 2).
3. **Preprocessing:** apply exhaustive resolution steps to eliminate variables, equality reduction, subsumption, etc.
4. **In processing**—apply the preprocessing simplifications at various points during solving.

Assumptions

- Assumptions. A useful technique is solving the formula **F** subject some set of literals called assumptions:

$$\mathbf{A} = \{l_1, l_2, \ldots, l_k\}$$

- The sat solver returns a truth assignment satisfying the formula **and** also making all assumptions literals true.
- If no such truth assignment exists it returns a clause

$$\mathbf{c} = (\neg l_1^c, \neg l_2^c, \ldots, \neg l_j^c)$$

Such that $\mathbf{F} \models \mathbf{c}$ and $l_i^c \in \mathbf{A}$.
This clause says that at least one of these literals must be false. (It specifies a subset of **A** that cannot be made true).

Assumptions

- **This is achieved by forcing the SAT solver to pick the assumption literals as its first set of decisions.**
- Initially every decision is the next unassigned literal in **A**, until here are no more unassigned literals in **A**.
- After assigning all literals in A we then continue the normal SAT solving process with the freedom to pick any decision variables we want.

Assumptions

- Either the sat solver finds a satisfying assignment (that makes **A** true) or it learns a clause **b** falsified by the levels containing the decisions over **A**.
- If we resolve away all forced literals in **b** to obtain an **all-decision** clause which is the clause **c** we want.
  - All decisions at and above the level **b** is falsified are assumption literals

MaxHS a hybrid approach to solving Maxsat

Fahiem Bacchus
University of Toronto

Outline

- General Purpose Exact optimizers
- MaxSat—an optimization version of SAT
- MaxHS
- Empirical Results

General Purpose Exact Optimizers

Discrete Optimization

- Discrete Optimization problems are ubiquitous in AI
  - Decision making problems with a payoff we want to maximize.
  - Problems that we want to solve that can be formulated as an optimization problems.

- Often these optimization problems are NP-Hard so a major challenge is to find solutions within feasible resource limits.
  - The resources available depend on the application.

Discrete Optimization

- Much work has been done on problems with special structure, e.g., convex, sub modular, bounded tree-width. This structure admits sophisticated analysis and often poly-time exact or approximation algorithms.

- However, not all problems have such structure.
- Often the theoretical approximation guarantees are weaker than needed in practice.

General Purpose Optimizers

- Exact general purpose optimizers, e.g., MIP solvers and more recently MaxSat solvers can be viable alternatives.
- The worst case complexity often makes people shy away from using such solvers.
- However such solvers are seen tremendous advances in performance in the past couple of decades, and in practice can often provide a better solution.

- Many industrial problems are solved with IP and SAT solvers.

General Purpose Optimizers

- The main attractive feature of such solvers is that they do not require that the input problem has any particular type of structure.
- So they can be applied to a wider range of problems, or applied to a more accurate model of the problem.

MaxSat via IP and SAT Solvers

- MaxSat is an optimization version of the SAT problem that can represent a range of optimization problems.
- In this talk I will discuss a hybrid solver for MaxSat that utilizes both SAT and IP solving.

MaxSat

The MaxSat Problem

- In theoretical studies MaxSat is taken to be the problem of satisfying a maximum number of clauses of a CNF formula.
- We can generalize this to associate a weight with each clause and make the problem be one of satisfying a maximum weight of clauses.
  - Equivalently MaxSat can be seen as a **minimization** problem: minimize the weight of the falsified clauses.
- This generalization is far more useful for modeling practical problems.

The MaxSat Problem

- Input:
  - a propositional formula in Conjunctive Normal Form
    - A conjunction of clauses
    - Each clause is a disjunction of literals
    - Each literal is a propositional variable or the negation of a propositional variable.
  - A cost (weight) associated with falsifying each clause

- Output:
  - **A MaxSat Solution:** a truth assignment of minimum cost
    - This truth assignment falsifies a minimum weight of clauses
    - Equivalently it satisfies a maximum weight of clauses.

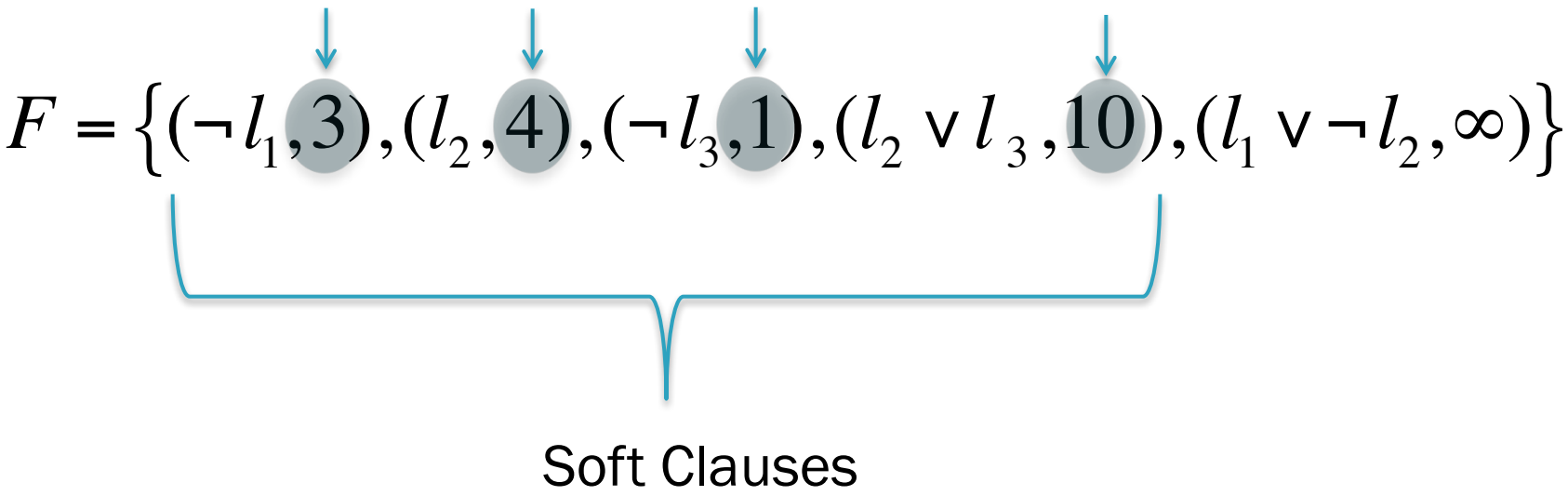
The MaxSat Problem

- If the weight of a clause is infinite then it costs an infinite amount to falsify it, i.e., it must be satisfied.
- Infinite weight clauses are called **hard** clauses, finite weight clauses are called **soft** clauses.

Max-SAT Example

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

Max-SAT Example



Max-SAT Example

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

Hard Clause

Max-SAT Example

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

l_1	l_2	l_3	Cost
0	0	0	

Max-SAT Example

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

l_1	l_2	l_3	Cost
0	0	0	4

Max-SAT Example

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

l_1	l_2	l_3	Cost
0	0	0	4 + 10

Max-SAT Example

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

l_1	l_2	l_3	Cost
0	0	0	14
0	0	1	5
0	1	0	∞
0	1	1	∞
⋮			

Max-SAT Example

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

l_1	l_2	l_3	Cost
0	0	0	14
0	0	1	5
0	1	0	∞
0	1	1	∞
⋮			

Max-SAT Solution

$$\text{cost}(l_1, l_2, \neg l_3) = 3$$

MaxSat

- Unweighted version is complete for the complexity class FP^{NP} of functions computable in polynomial time given access to an NP oracle
- APX-Complete (no polynomial time approximation scheme unless P=NP)
- Many important problems fall into this class and can therefore be efficiently expressed as MaxSat
- Bioinformatics, Electronic Design Automation, Operations Research, various planning problems…

- The MaxSat encoding is often quite natural.

Example, correlation clustering

- A collection of objects that we wish to partition into clusters of similar objects.
- Represent the objects as vertices in a graph.
- Each edge has a weight---negative if the objects it connects are not similar, positive if the objects are similar.

- Goal: Partition the vertices so that the following sum is minimized:
  - weight(e) for every edge e with positive weight where its two vertices are in different clusters
  - -weight(e) for every edge e with negative weight where its two vertices are in the same cluster.

Example, correlation clustering

- Unknown number of clusters, if we have n objects we can have n clusters (0—n-1).
- For each object use $\log_2(n)$ propositional variables whose F/T (0/1) values represent the base 2 encoding of that object's cluster.
- Two objects o1, o2 with an edge between them are in the same cluster if propositional variable $s_{1,2}$ is true.

  $s_{1,2}$ ⇔ all bits in the bit encodings are the same

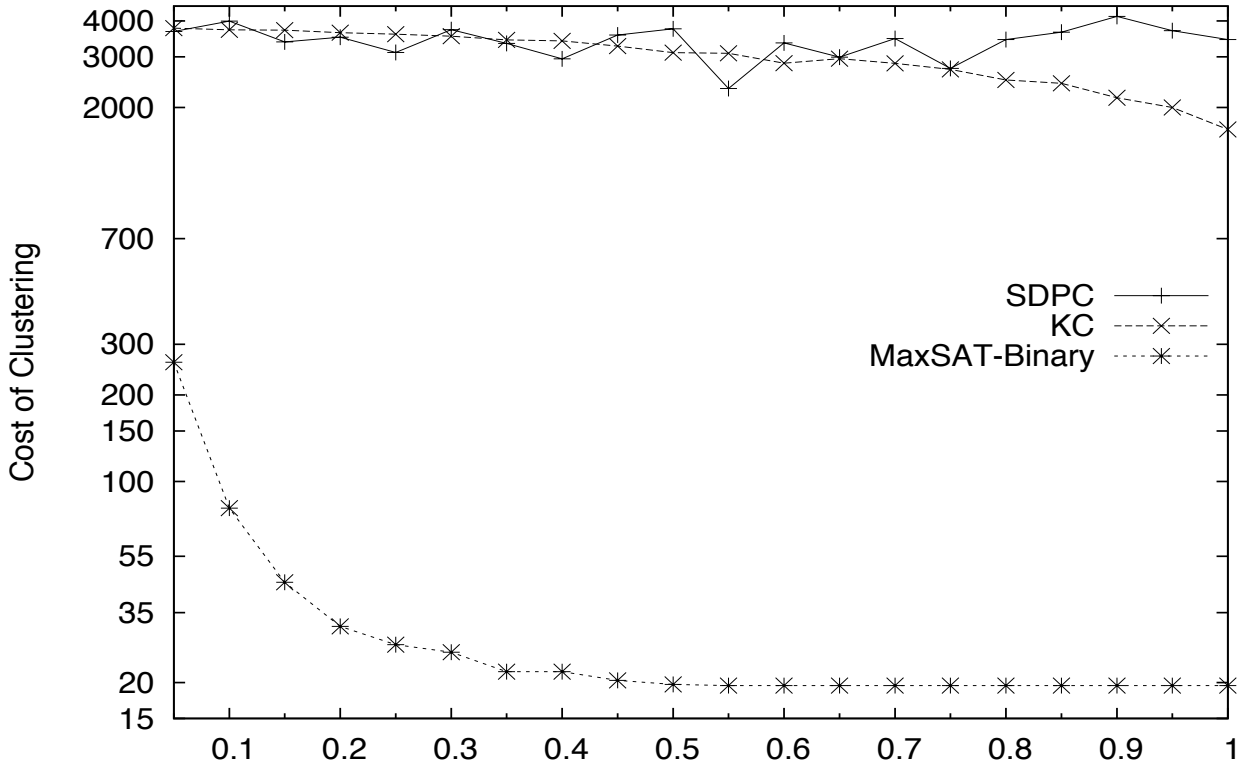
  a set of hard clauses.
- Soft clauses, for every pair of objects o1, o2, where wt is the weight of the edge between these objects:

  $(\neg s_{1,2}; wt)$ when $wt > 0$

  $(s_{1,2}; -wt)$ when $wt < 0$

Example, Correlation Clustering Approximation quality

[Berg and Järvisalo, 2016]



Solving times 10-100 seconds.

x-axis degree of completeness of the graph

- SDPC—approximation based on rounding a semi-definite program
- KC—greedy approximation

Reported Applications of MaxSat

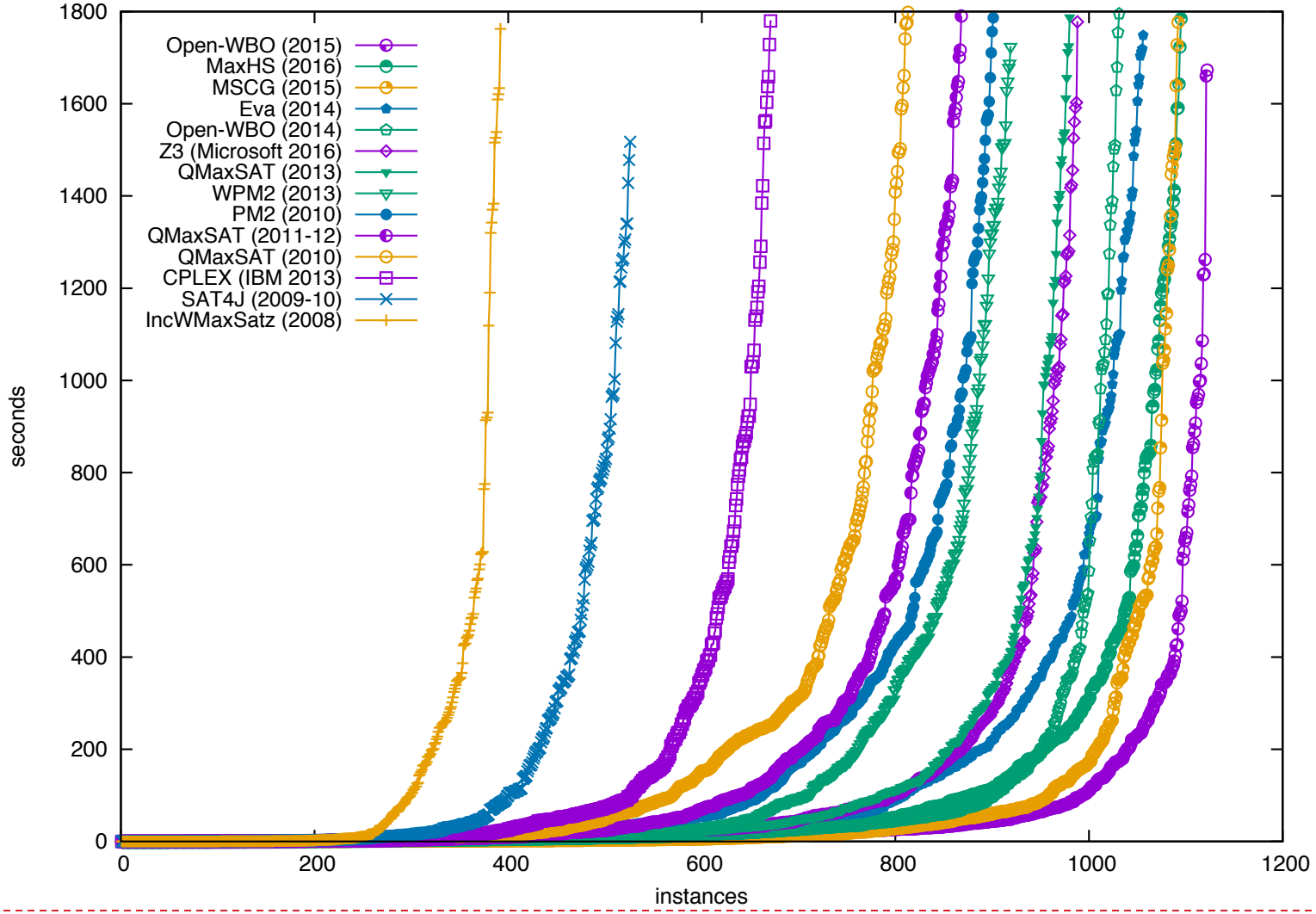
probabilistic inference	[Park, 2002]
design debugging	[Chen, Safarpour, Veneris, and Marques-Silva, 2009]
	[Chen, Safarpour, Marques-Silva, and Veneris, 2010]
maximum quartet consistency	[Morgado and Marques-Silva, 2010]
software package management	[Argelich, Berre, Lynce, Marques-Silva, and Rapicault, 2010]
	[Ignatiev, Janota, and Marques-Silva, 2014]
Max-Clique	[Li and Quan, 2010; Fang, Li, Qiao, Feng, and Xu, 2014; Li, Jiang, and Xu, 2015]
fault localization	[Zhu, Weissenbacher, and Malik, 2011; Jose and Majumdar, 2011]
restoring CSP consistency	[Lynce and Marques-Silva, 2011]
reasoning over bionetworks	[Guerra and Lynce, 2012]
MCS enumeration	[Morgado, Liffiton, and Marques-Silva, 2012]
heuristics for cost-optimal planning	[Zhang and Bacchus, 2012]
optimal covering arrays	[Ansótegui, Izquierdo, Manyà, and Torres-Jiménez, 2013b]
correlation clustering	[Berg and Järvisalo, 2013; Berg and Järvisalo, 2016]
treewidth computation	[Berg and Järvisalo, 2014]
Bayesian network structure learning	[Berg, Järvisalo, and Malone, 2014]
causal discovery	[Hyttinen, Eberhardt, and Järvisalo, 2014]
visualization	[Bunte, Järvisalo, Berg, Myllymäki, Peltonen, and Kaski, 2014]
model-based diagnosis	[Marques-Silva, Janota, Ignatiev, and Morgado, 2015]
cutting planes for IPs	[Saikko, Malone, and Järvisalo, 2015]
argumentation dynamics	[Wallner, Niskanen, and Järvisalo, 2016]

Reported Applications of MaxSat

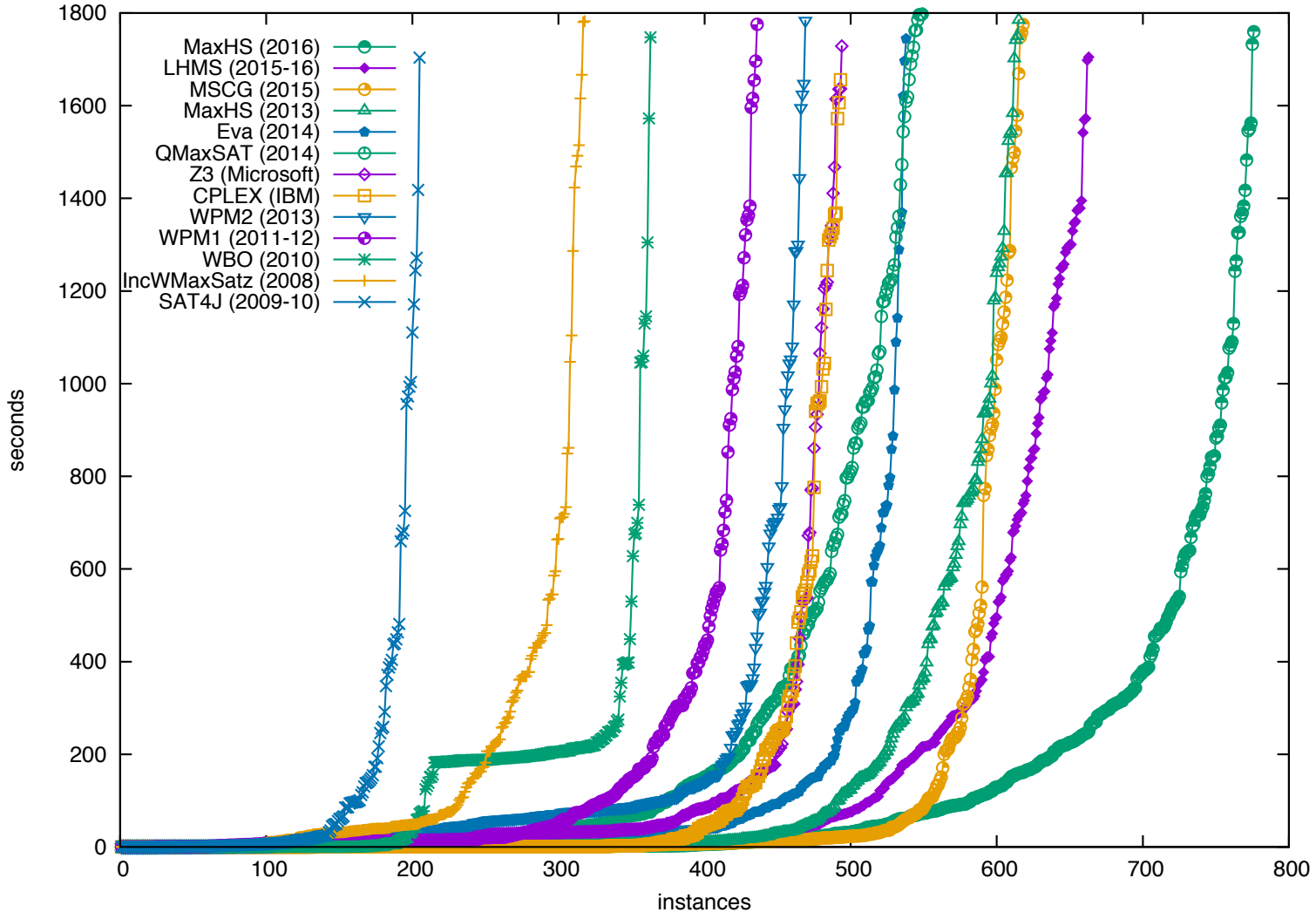
- Growing number of applications being reported in the last couple of years.

- Advances in MaxSat Solver technology are central to this increasing success

Improvements in MaxSat Solving UNWEIGHTED (2008-2016)



Improvements in MaxSat Solving WEIGHTED (2008-2016)



Problem Sizes

- Largest problems solved in MaxSat Evaluation, >6,000,000 variables and > 13,000,000 clauses (solved by MaxHS in < 800 sec.)
- MaxSat is considerably harder than SAT, for SAT problems as big as >10,000,000 variables and >50,000,000 clauses have been solved.

MaxHS

Prior Methods for Solving MaxSat

- Most MaxSat solvers exploit a SAT solver to solve a series of SAT decision problems
- Relaxation/blocking variables are used to control which clauses must be satisfied.

Relaxation Variables

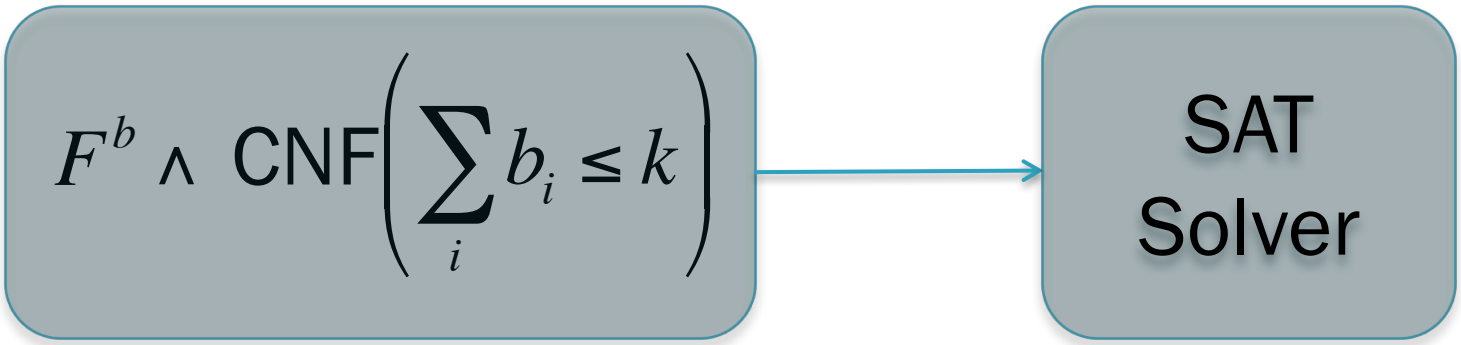
$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

$$F^b = \{(\neg l_1 \vee b_1), (l_2 \vee b_2), (\neg l_3 \vee b_3), (l_2 \vee l_3 \vee b_4), (l_1 \vee \neg l_2)\}$$

- Add a fresh variable b_i to each soft clause
- Drop the clause weights
- F^b is satisfiable if hards are satisfiable, since setting b_i to true removes the original soft clauses

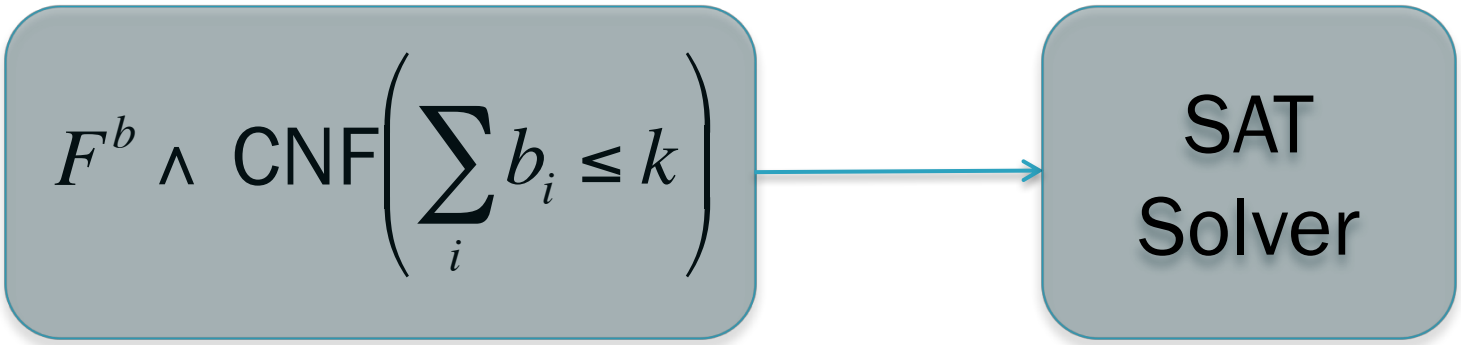
Basic Sequence of SAT

- Suppose all soft clauses have weight 1
- Add a cardinality constraint over the relaxation variables, limiting how many can be assigned to True (i.e., how many softs can be falsified)



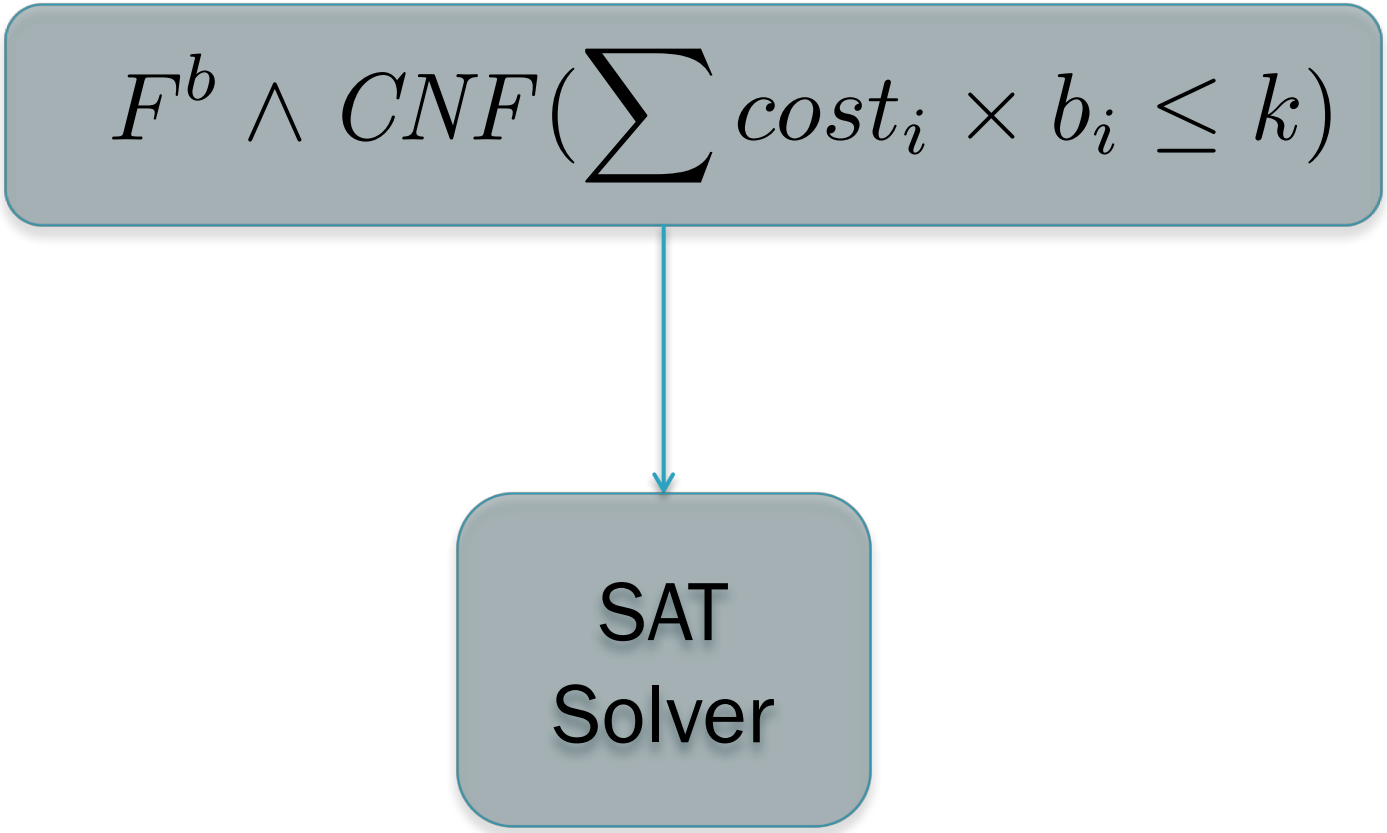
Basic Sequence of SAT

- Observation: If *k* is the *minimum* number of softs that can be falsified then the formula is satisfiable, and each satisfying solution is a is a Max-SAT solution



Basic Sequence of SAT

- This approach can be extended to non-uniform weights



Basic Sequence of SAT

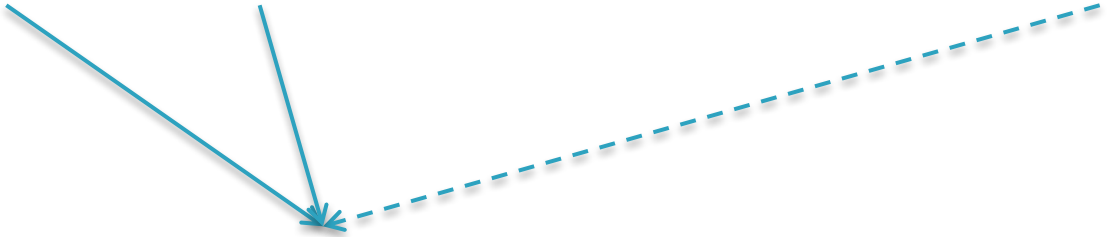
- Can no longer use simple cardinality constraints
- One has to encode linear equations over the b-variables into CNF to capture the different costs (pseudo-boolean constraints).
- Such constraints are hard for the SAT solver.
- Even for the unweighted case the sum over all b-variables requires a very large and inefficient encoding—when you have thousands/millions of soft clauses.

Core Based SAT approaches

- This simple approach can be significantly improved by utilizing cores.

Cores

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

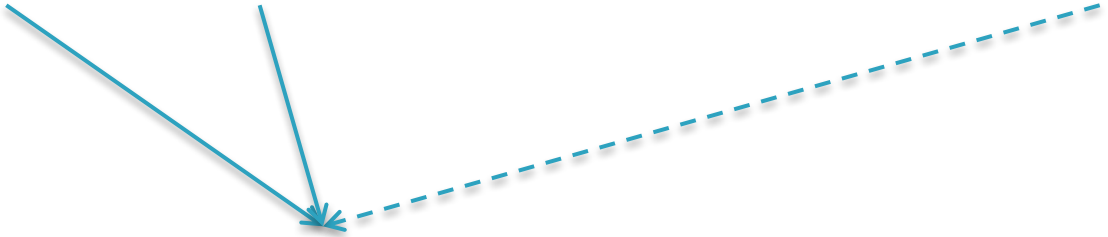


Core: $K_1 = \{(\neg l_1, 3), (l_2, 4)\}$

- A core is a set of soft clauses that is inconsistent with the hard clauses

Cores

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$



Core: $K_1 = \{(\neg l_1, 3), (l_2, 4)\}$

- Using {-bi | bi is a blocking variable} as assumptions SAT solvers can return a core.

Cores

- Assumptions are a set if literals. When given assumptions A the SAT solver will either find a satisfying model in which every literal in A is true
- Or it will return a clause containing only negated literals of A.
- So when A = {-bi | bi is a blocking variable} the returned clause will be of the form
  (b1, b3, b5, b6, …) a set of positive b-literals
- bi = True ➔ soft clause ci is falsified. So this clause specifies a subset of soft clauses at least one of which must be falsified: a core.

Core Based MaxSat Algorithms

- Observation: at least one of the clauses in a core will be falsified by the Max-SAT solution
- Idea: given a core, we can use cardinality constraints over **only** the relaxation variables of the soft clauses in the core to express this condition.
- These are typically much smaller than cardinality constraints over all relaxation variables.

Core Based MaxSat Algorithms

- The cardinality constraint “relaxes” the formula…it allows one of these soft clauses in the core to be falsified.
- If that relaxation is insufficient another core will be found, and the formula can be further relaxed.

Core Based MaxSat Algorithms

- Today, many modern MaxSat solvers such as RC-2, WBO, and Eva500 are based on this idea.
- Works well when
  1. Very few soft clauses are falsified in the optimal model (< 200)
  2. Very small number of distinct clause weights (< 3)

The MaxHS Approach

Motivation

- Existing MaxSat solvers suffer because they create harder and harder SAT problems by adding cardinality constraints over the b-variables.
- The situation is worse when soft clauses have diverse weights--SAT solvers are not very good at dealing with pseudo Boolean constraints.
  - SMT-solvers don't offer any significant improvement
- MaxSat problems can also be converted to an Integer Program. But IP solvers perform poorly because the linear constraints arising from the clauses often yields a poor linear relaxation.

The MaxHS Approach

- The SAT problems are subsets of the original Max-SAT formula
  - They are likely to be no harder for a SAT solver than the original formula
- All numeric reasoning about costs is delegated to an Integer Programming solver (CPLEX)
  - designed for optimization
  - costs can be floating point numbers
  - the underlying LP + Cuts approach is very powerful

Multiple Cores

- A core says that at least one of the soft clauses in it must be falsified
- Idea: generalize this observation to multiple cores

- [Bacchus, Cho, Davies 2010, Helmert & Bonet 2010]

Hitting Sets

Cores

$$K_1 = \{C_1, C_3, C_{10}\}$$
$$K_2 = \{C_3, C_4\}$$
$$K_3 = \{C_5, C_9, C_{11}\}$$

- A *hitting set* is a subset of the soft clauses, that includes at least one clause from each core

Hitting Sets

Cores

$$K_1 = \{C_1, C_3, C_{10}\}$$
$$K_2 = \{C_3, C_4\}$$
$$K_3 = \{C_5, C_9, C_{11}\}$$

$$hs_1 = \{C_3, C_5\}$$
$$cost(hs_1) = wt(C_3) + wt(C_5)$$

- A hitting set is a subset of the soft clauses, that includes at least one clause from each core
- We are interested in hitting sets of minimum cost

Cores and hitting sets

- Remember
  - A set of soft clauses $\boldsymbol{\kappa} \subseteq \mathbf{S}$ is a **core** of **F** if $\boldsymbol{\kappa} \cup \mathbf{H}$ is **UNSAT**
  - Feasible solutions satisfy the hard clauses **H**
- Let **K** be any set of cores of F and **π** any feasible solution. **π must falsify** at least one soft clause of every core in **K.**
- Let A = {c | $\pi \not\models$ c} be the set of clauses falsified by π
- Then A is a hitting set of **K** (non-empty intersection with every member of **K**).

Cores and hitting sets

- Let MCHS(K) be a minimum cost hitting set of **K**–this is a set of soft clauses.
- For every feasible solution π

$$\text{cost}(\pi) = \text{wt}(A) \geq \text{wt}(\text{MCHS}(K))$$

- The weight of a minimum cost hitting set of **any** set of cores is a **lower bound** on the cost of an optimal solution.
- Therefore, for **any** set of cores **K** and **any** feasible solution π if cost(π) = wt(MCHS(K)), π must be an **optimal solution.**

- This leads to a simple algorithm for finding an optimal solution.

MaxHS Theorem

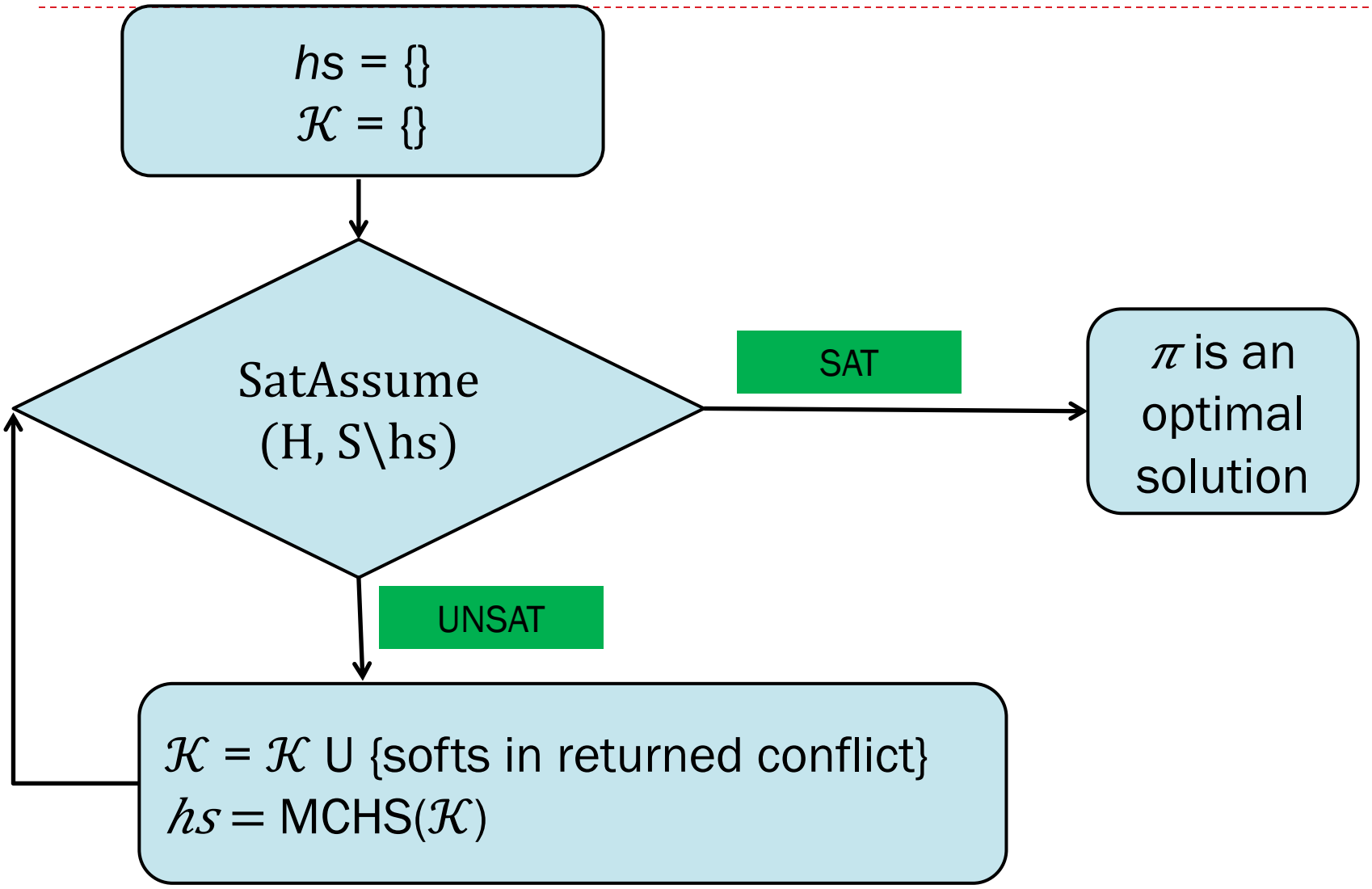
Theorem

If a truth assignment π satisfies $F \setminus hs$ where hs is a minimum cost hitting set of a collection of cores, then π is a Max-SAT solution.

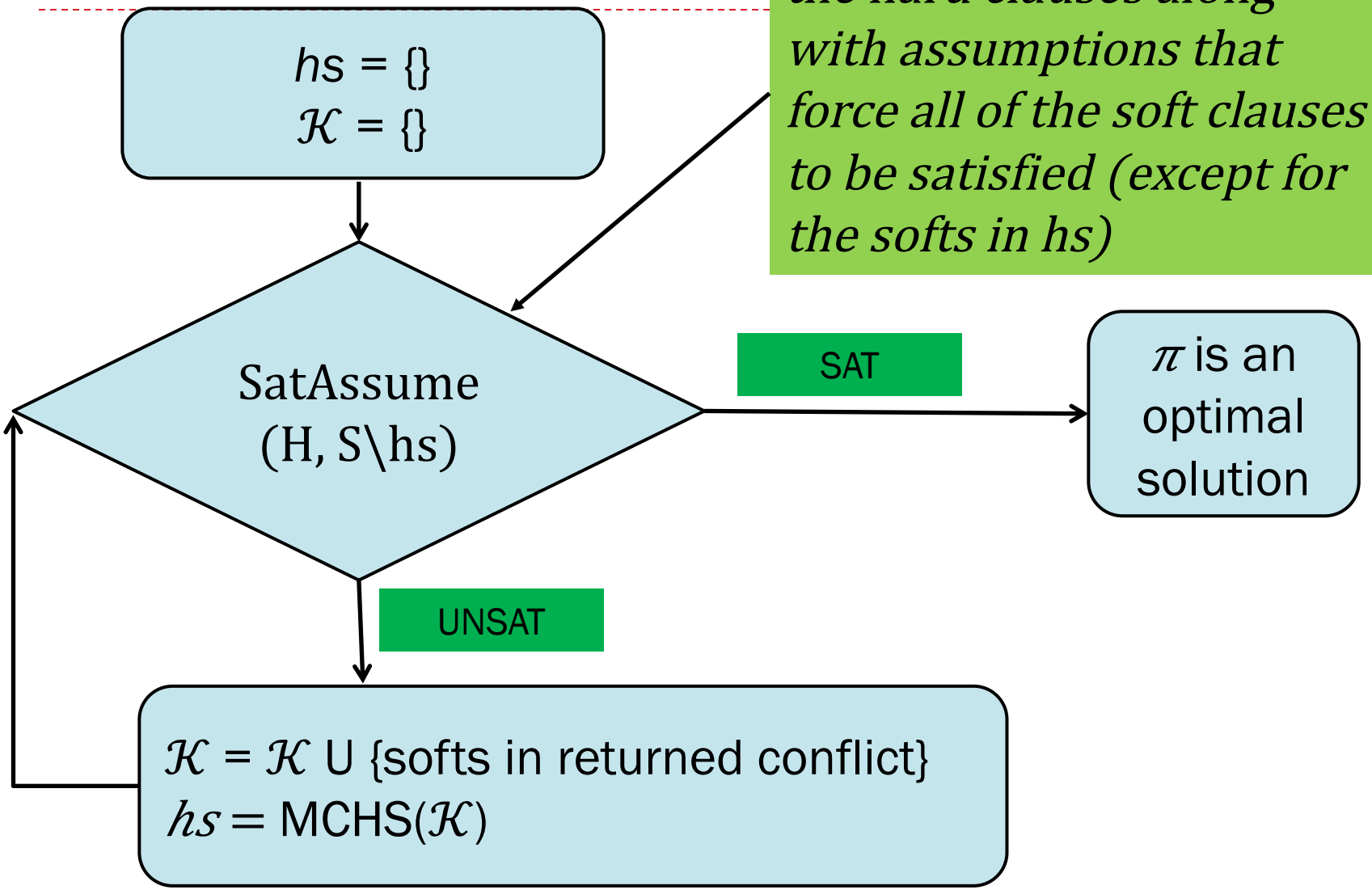
Proof Sketch: π has cost at most cost(*hs*) since it satisfies all clauses not in *hs*. But cost(*hs*) is a lower bound on the cost of the Max-SAT solution.

$$\text{cost}(\pi) \leq \text{cost}(hs) \leq \text{mincost}(F) \leq \text{cost}(\pi)$$

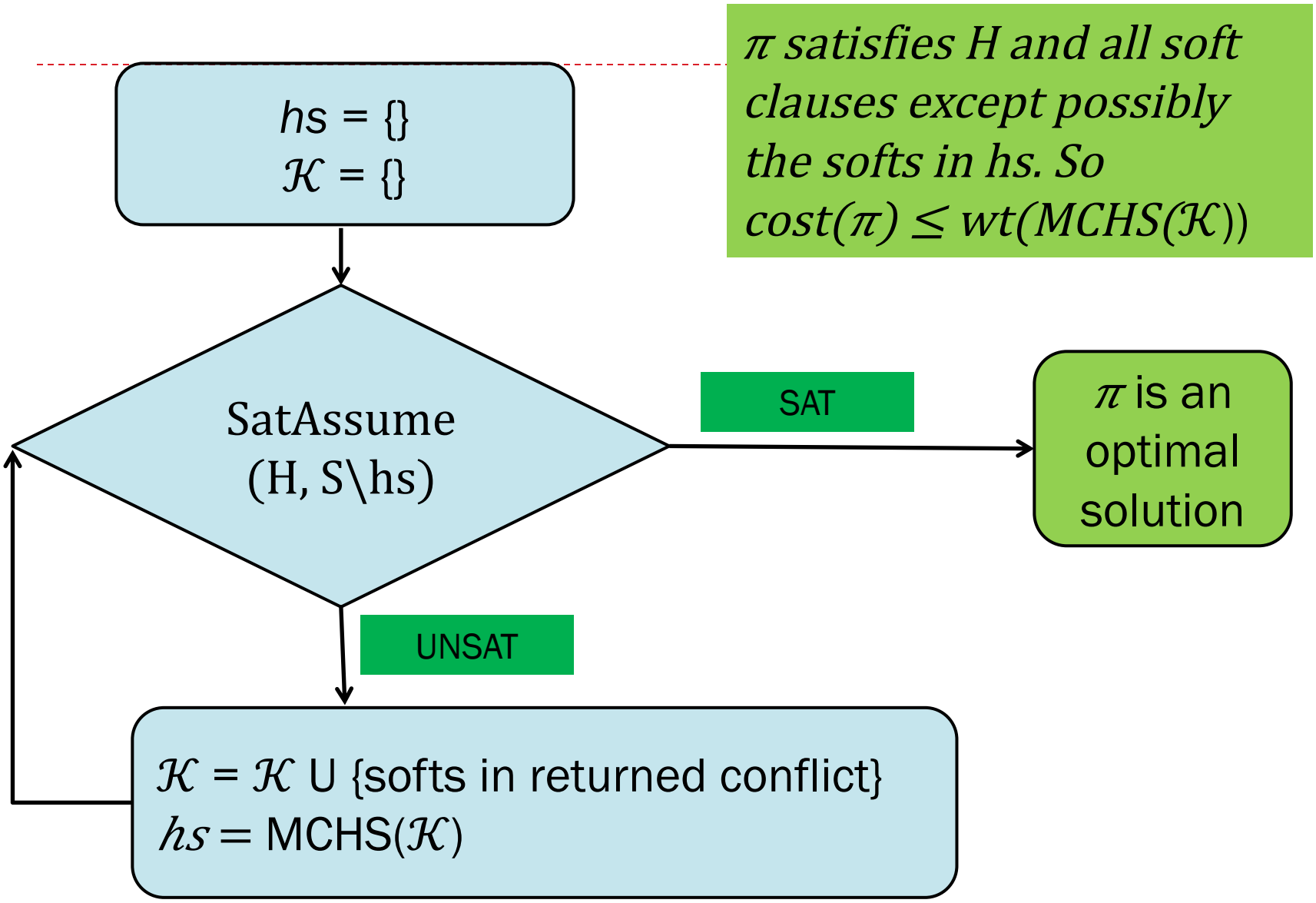
The IHS Algorithm



The IHS Algorithm

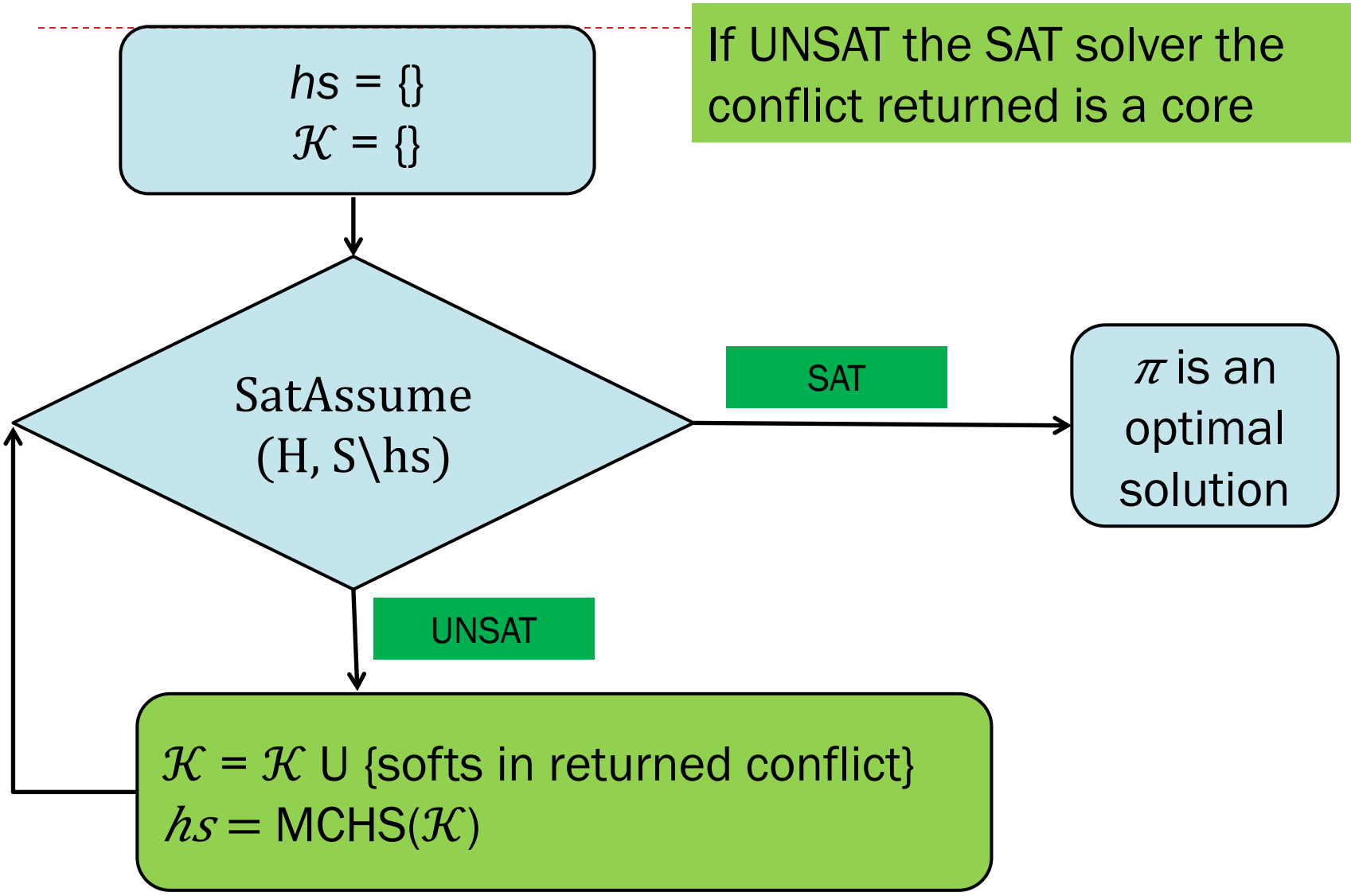


The IHS Algorithm



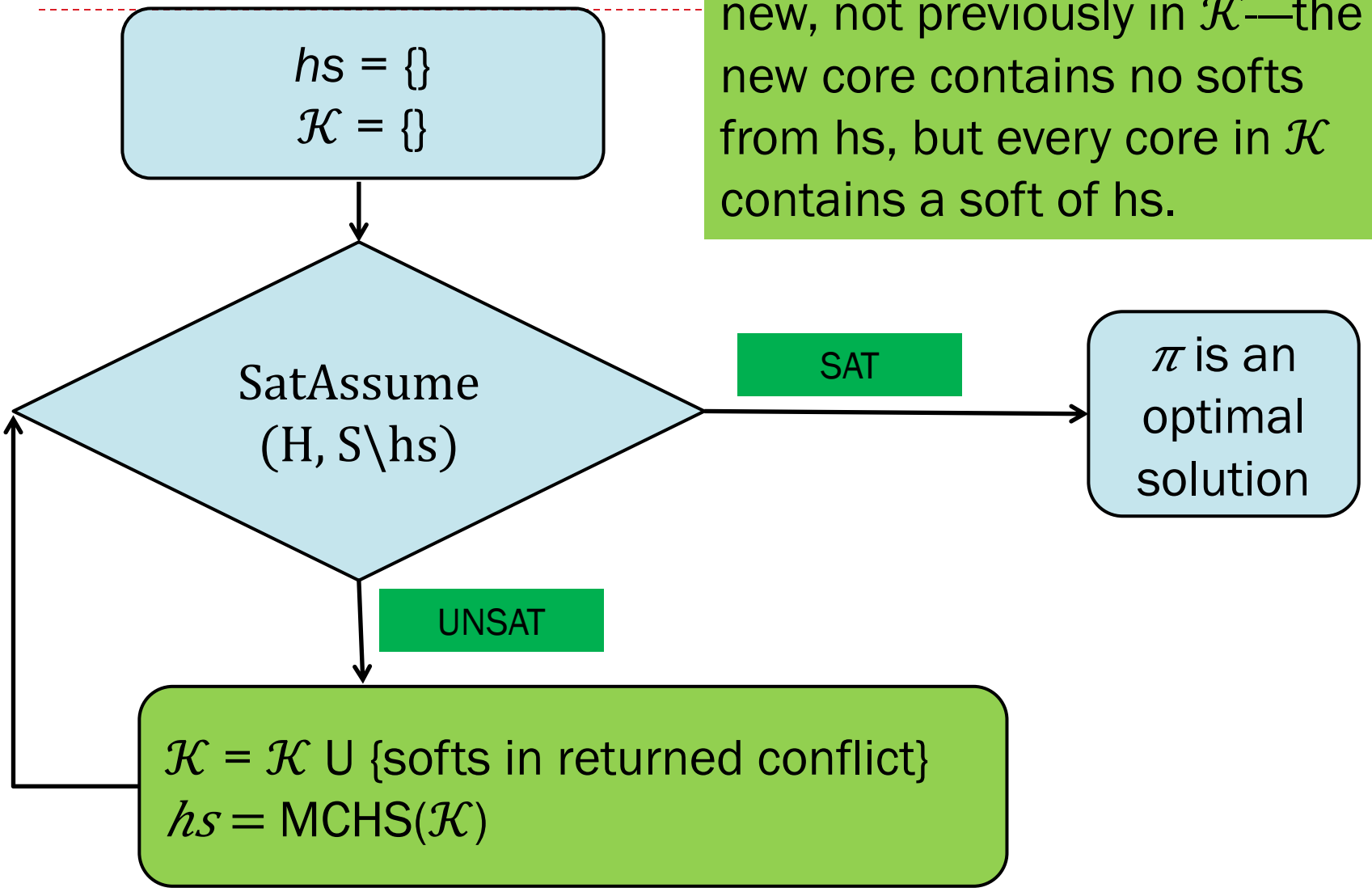
π satisfies H and all soft clauses except possibly the softs in hs. So $cost(\pi) \leq wt(MCHS(\mathcal{K}))$

The IHS Algorithm



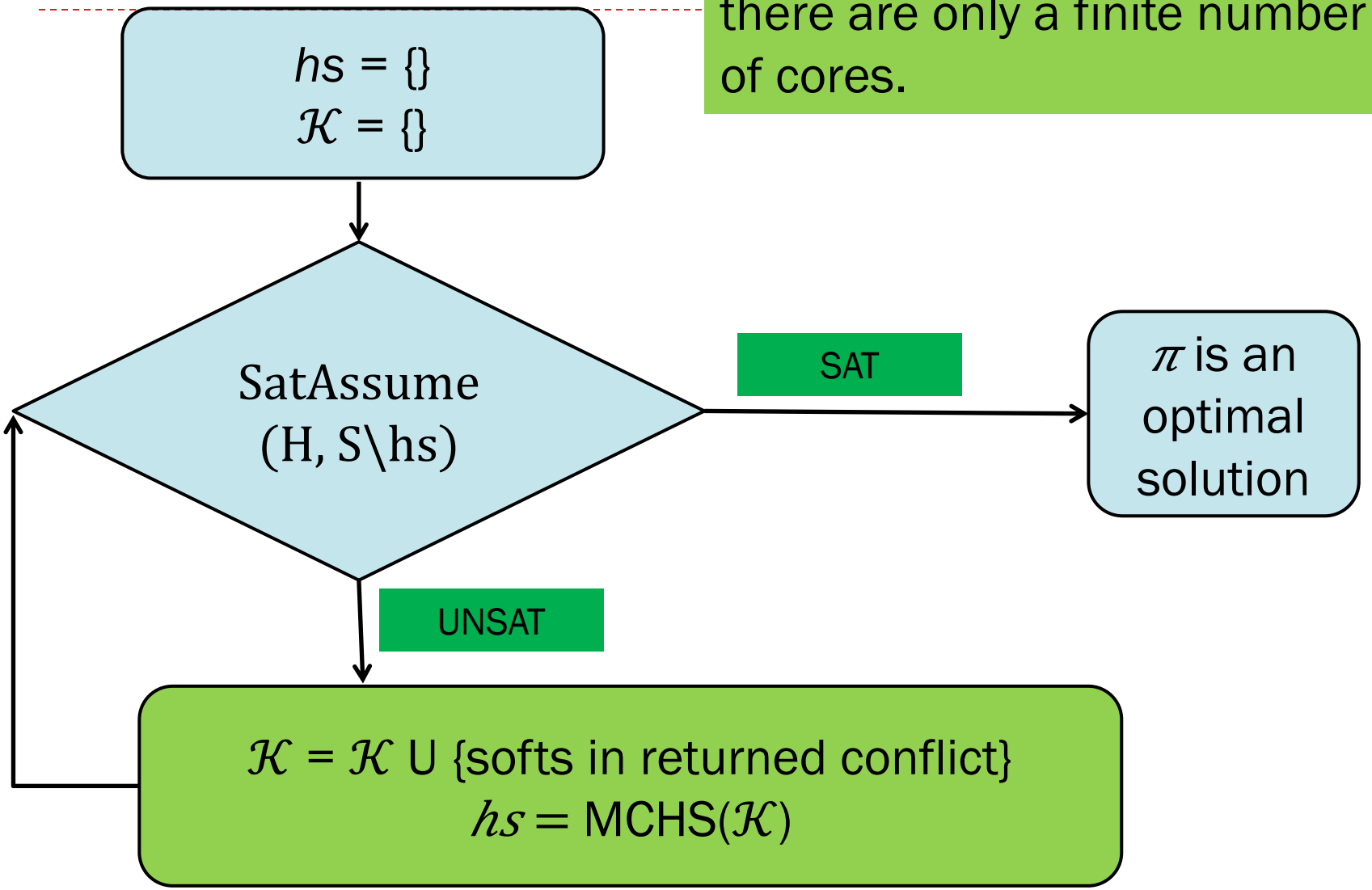
The IHS Algorithm

The returned core must be new, not previously in $\mathcal{K}$—the new core contains no softs from hs, but every core in $\mathcal{K}$ contains a soft of hs.



The IHS Algorithm

This process must terminate as there are only a finite number of cores.



MaxHS

- MaxHS is using SAT reasoning to incrementally construct an IP problem from the input MaxSat problem.
- If the set of soft clauses {c_1, c_4, c_6, c_7} is a core, the IP will contain the linear constraint

$$b_1 + b_4 + b_6 + b_7 \geq 1$$

  where the b_i are the clause relaxation variables indicating that at least one of them is true (=1).
- These constraints specify a hitting set (set-cover) problem.

MaxHS

- Set cover is hard in general, but MIP solvers like CPLEX are quite effective on set-cover.

- The approach is related to logic based Benders (Hooker). Also to the implicit hitting set formalism of Karp.

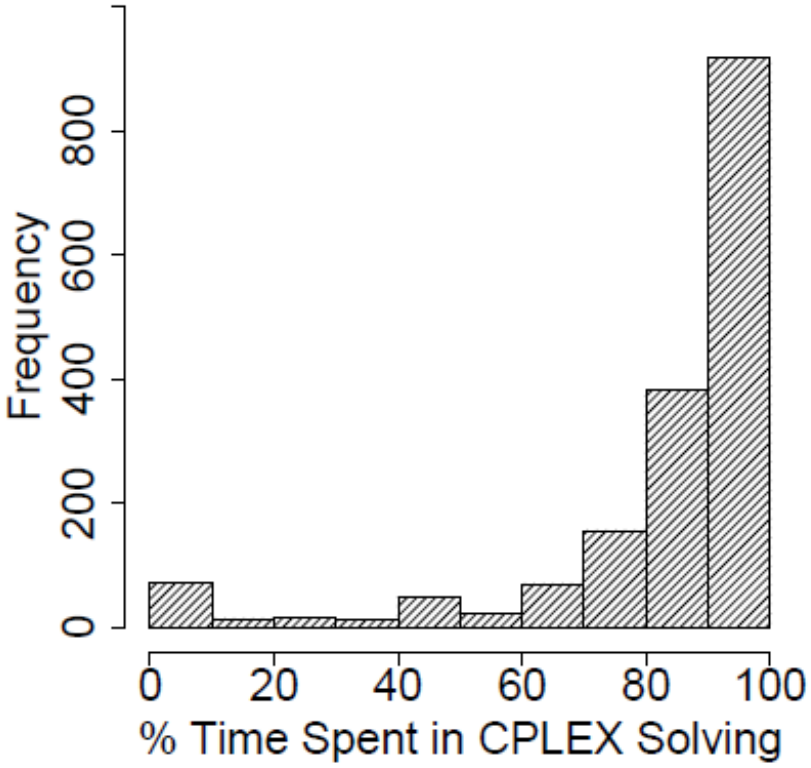
- This re-encoding can be much more effective than directly trying the solve the MaxSat problem with an IP solver

Behavior of MaxHS

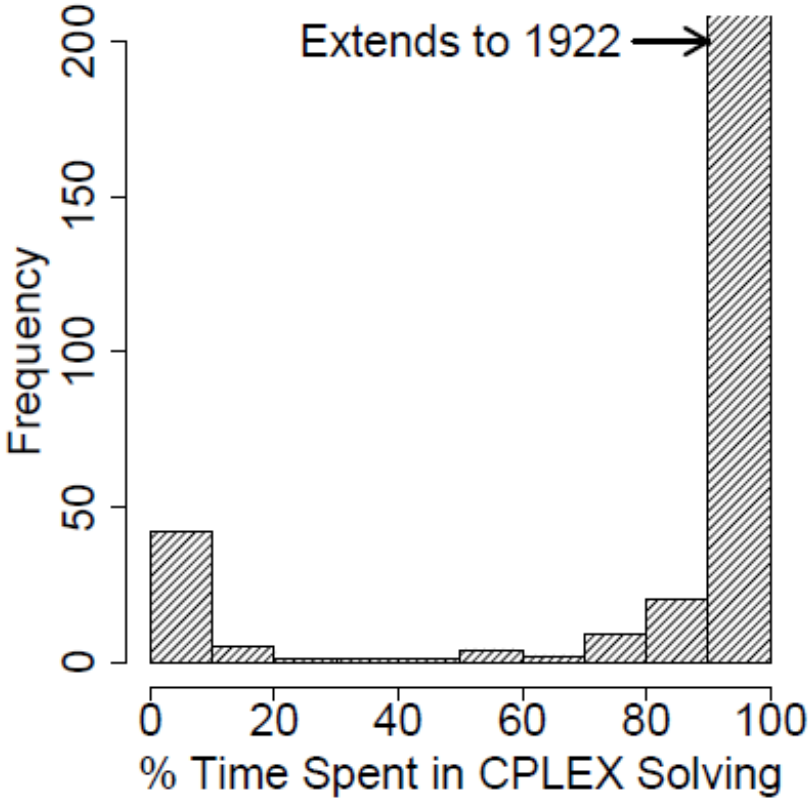
- MaxHS is incremental
  - Every iteration produces a lower bound on the MaxSat solution

- Three potential sources of exponential behaviour:
  1. SAT Solving
  2. Solving the NP-Hard minimum hitting set problem
  3. Number of iterations of SAT solving/hitting set computations

Where is the time spent?



Solved



Unsolved

Improving the basic algorithm

Solving the MCHS problem after every single core is too slow

1. Give better constraints to CPLEX
2. Generate constraints more cheaply without an expensive MCHS computation
3. Give CPLEX multiple constraints at a time thus reducing the total number of calls to CPLEX

Better Constraints: Minimal Cores

- The cores returned by the SAT solver may contain irrelevant clauses
- A *minimal* core is one for which no proper subset is a core
  - How to find minimal cores?
    - a simple algorithm that tests if each clause can be removed from the core with a call to the SAT solver.
    - There are improved algorithms for minimizing cores (Bacchus & Katsirelos CAV-2015). Some of these ideas have been exploited in MaxHS.
  - Finding minimal cores is too expensive, but we can spend some time making them smaller with these algorithms.

Cheaper Constraints: Seeding

- By examining the input CNF we can find constraints that can be feed directly into CPLEX.
- *Seed* CPLEX with a collection of such constraints, as a preprocessing step

Relaxation Variables

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

$$F^b = \{(\neg l_1 \vee b_1), (l_2 \vee b_2), (\neg l_3 \vee b_3), (l_2 \vee l_3 \vee b_4), (l_1 \vee \neg l_2)\}$$

- Don't really need a relaxation variable for unit soft clauses.
- E.g., l_1 indicates that the soft clause $(\neg l_1, 3)$ is falsified.

Cheaper Constraints: Equivalence Seeding

- Examine input formula for clauses all of whose variables appear in unit soft clauses.

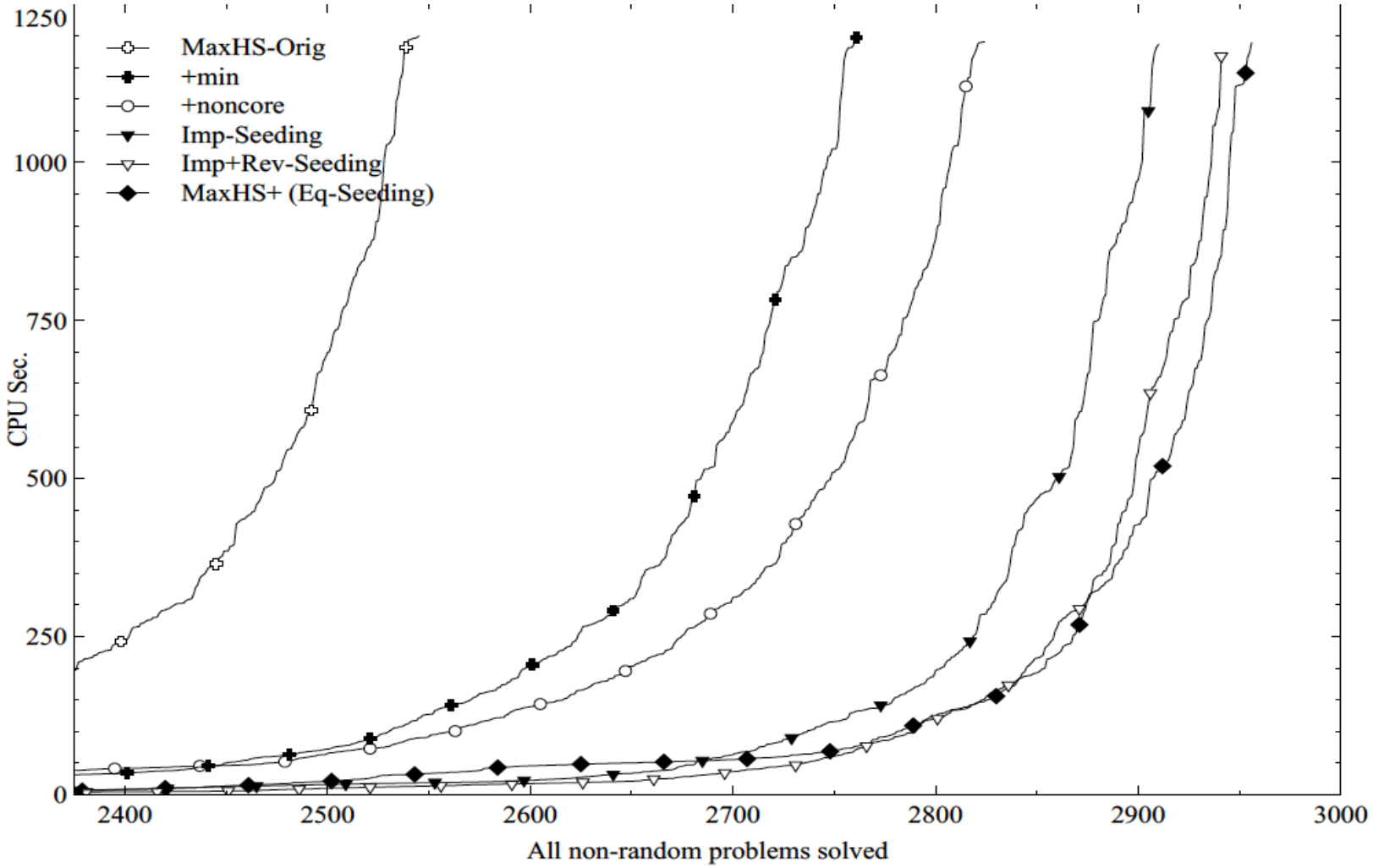
$(l_1, 3)\ (l_2, 2)\ (l_3, 10)$ $\qquad$ $(l_1 \vee \neg l_2 \vee \neg l_3)$

- The constraint

$$l_1 + (1 - l_2) + (1 - l_3) \geq 1$$

can be added to the IP solver.
Note that now the IP solver is not solving a pure set-cover problem. It is finding a constrained hitting set.

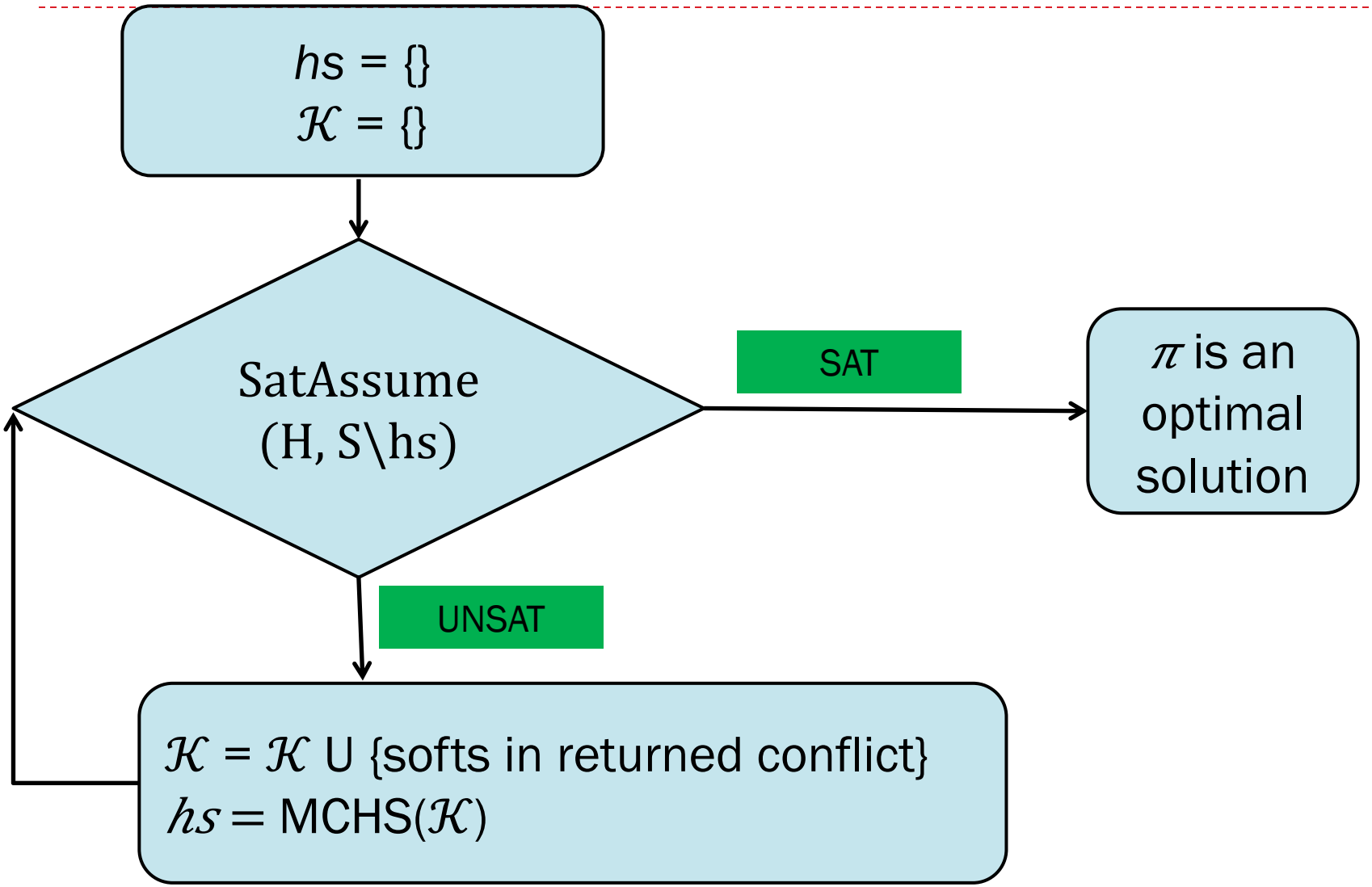
Experimental Results For these improvements



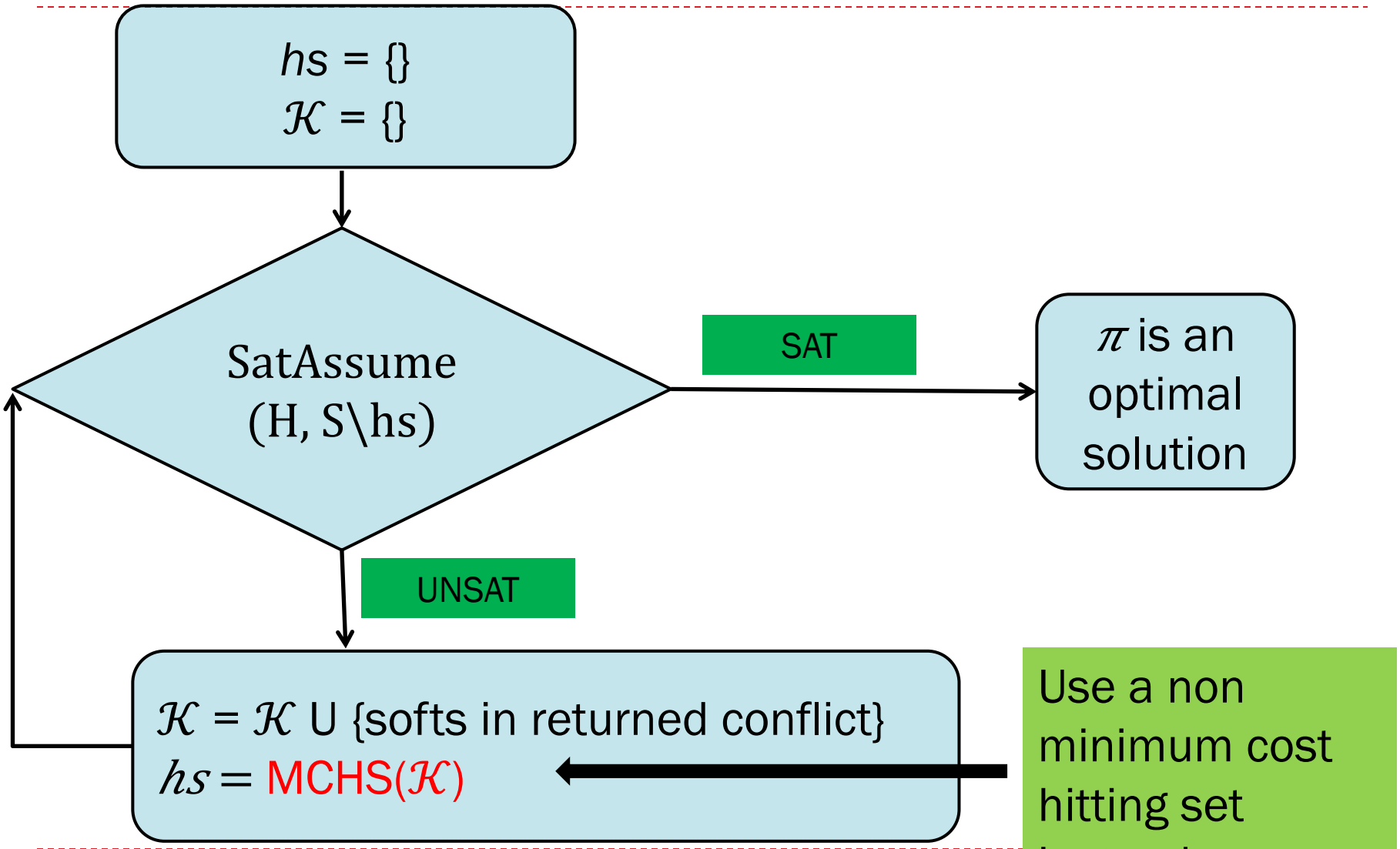
Multiple Constraints: Non-Optimal Hitting Sets

- At each iteration, a single constraint is added to the IP model and the hitting set problem is solved to optimality again
- Goal: reduce the number of times the hitting set problem must be solved to optimality.
- Use heuristics to find a non-optimal hitting set instead of an optimal one.

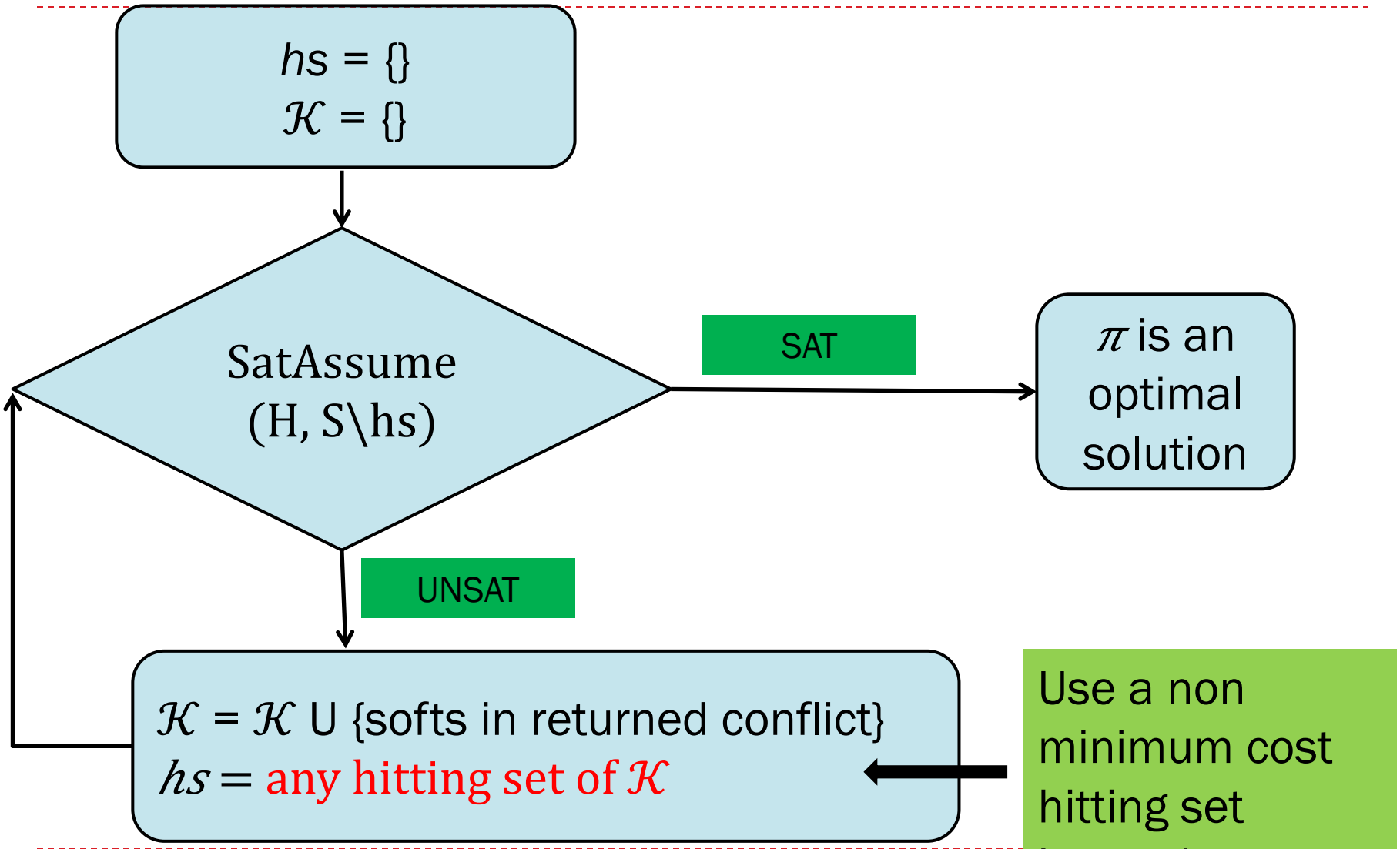
The IHS Algorithm



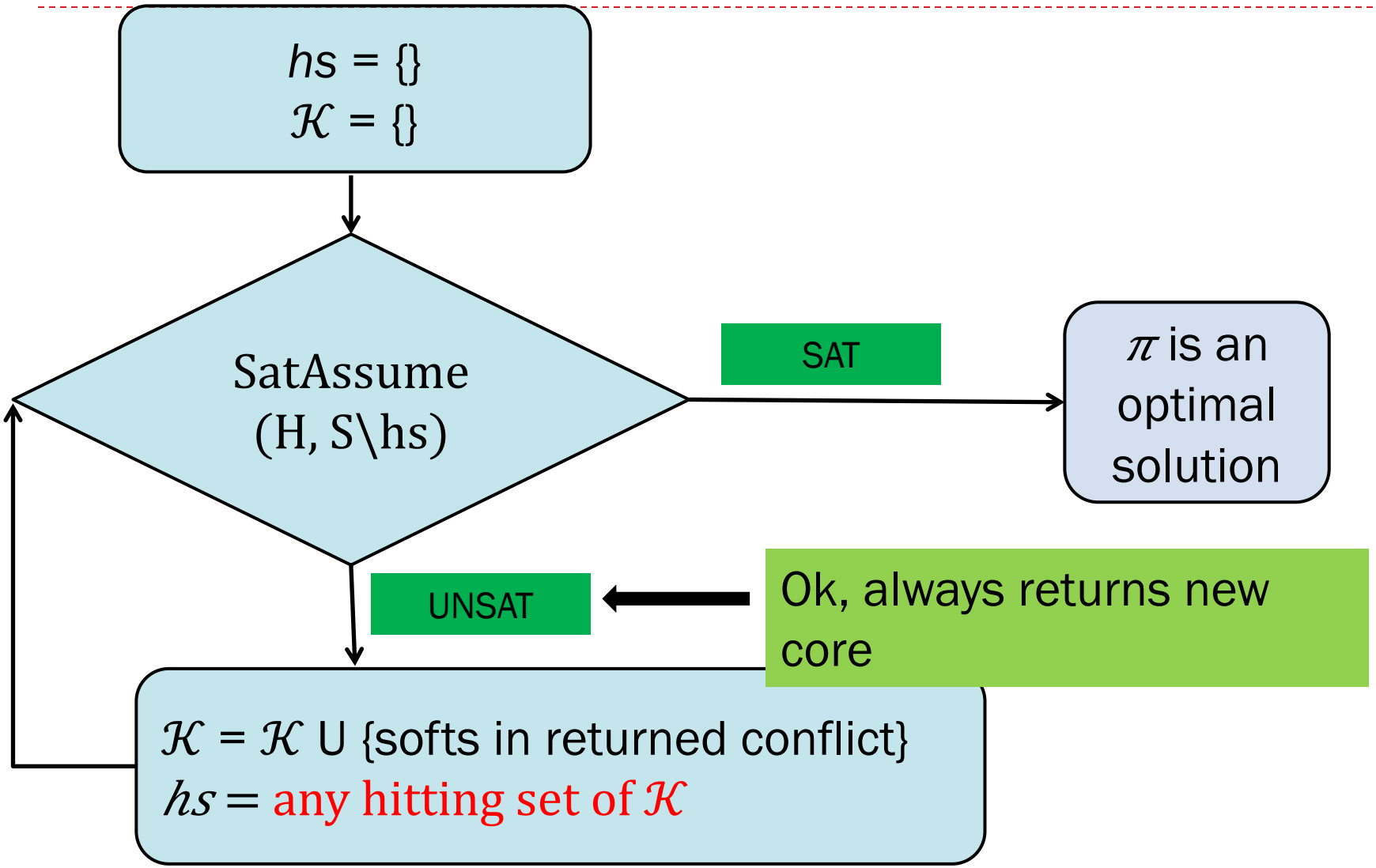
The IHS Algorithm



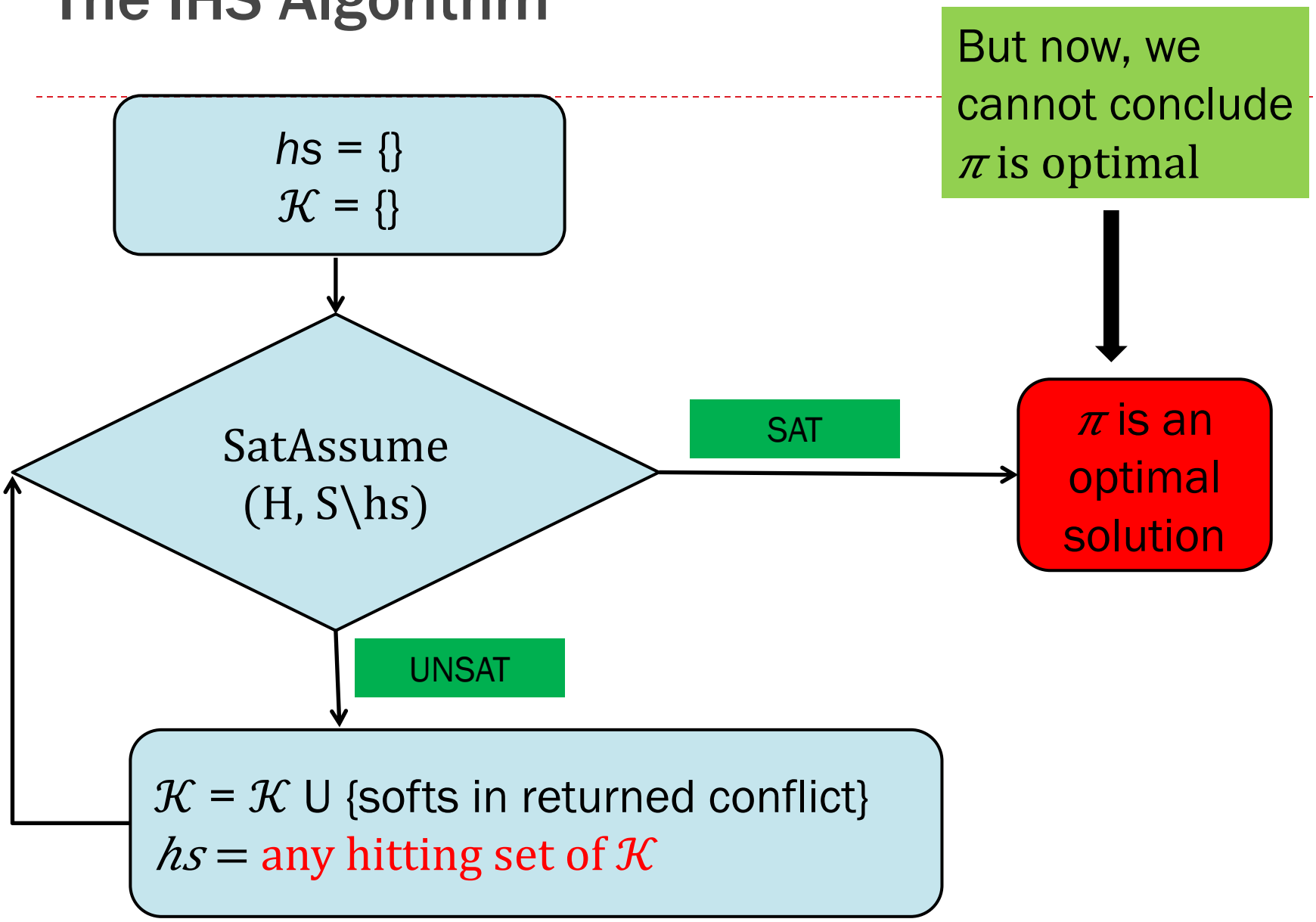
The IHS Algorithm



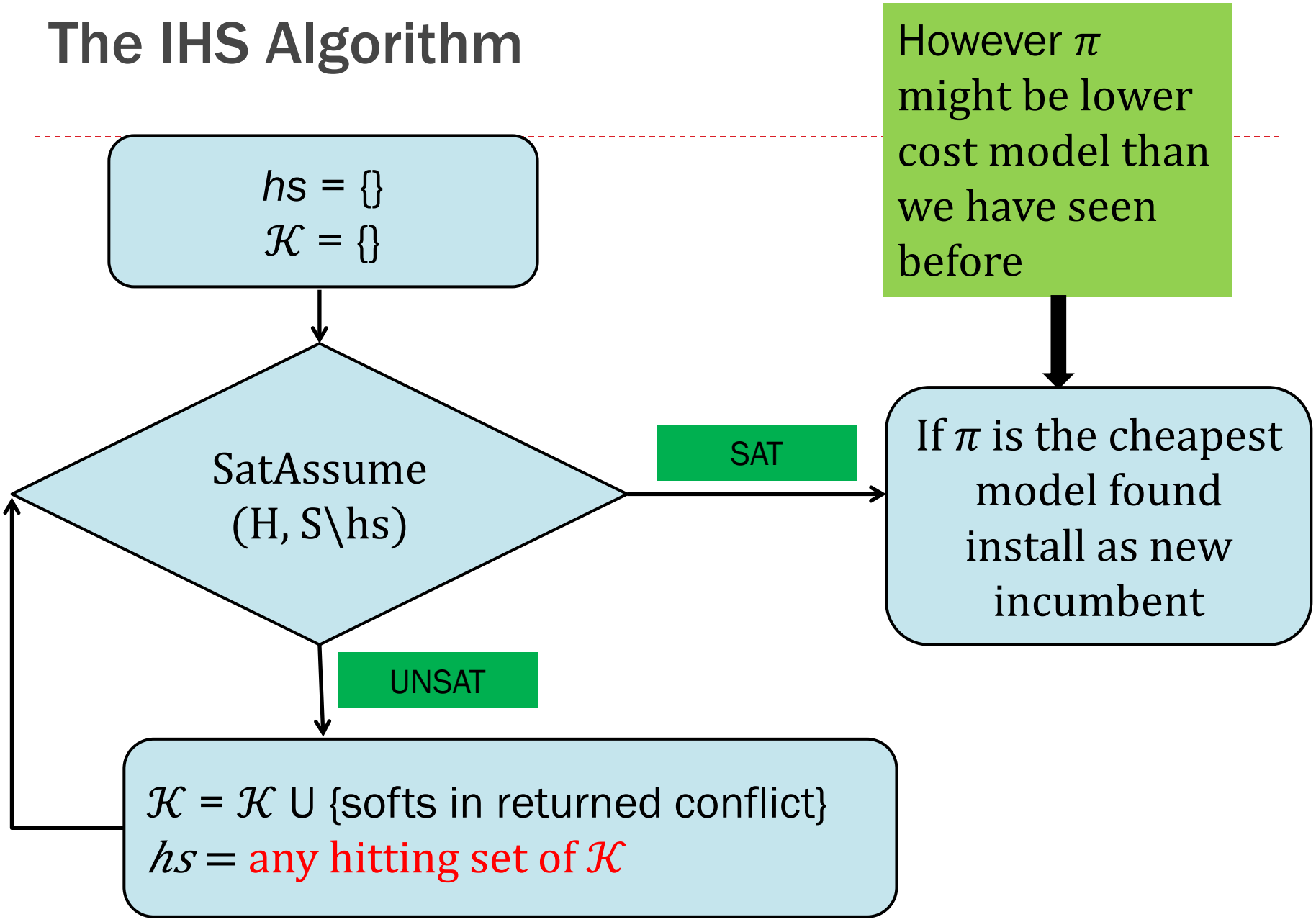
The IHS Algorithm



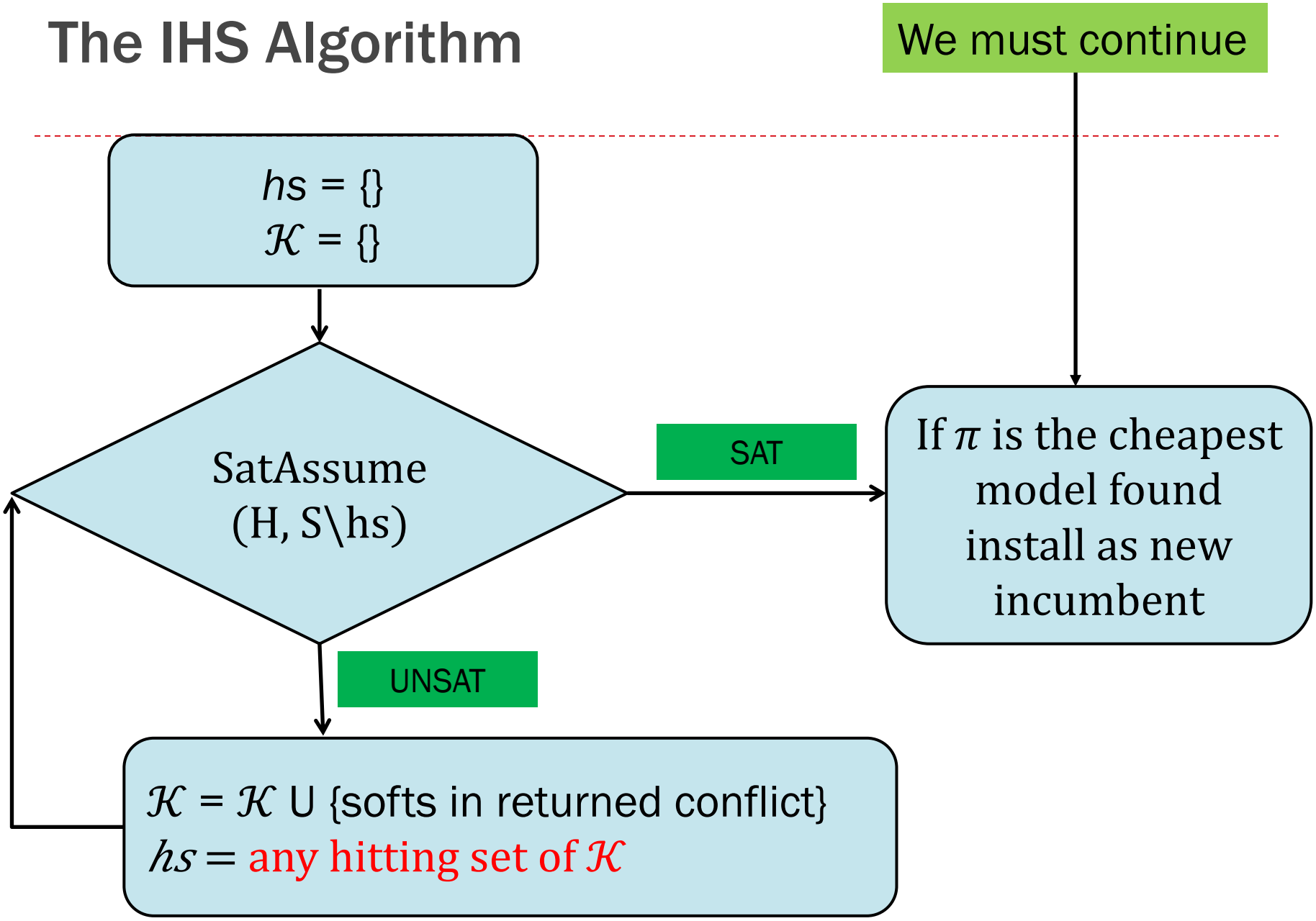
The IHS Algorithm



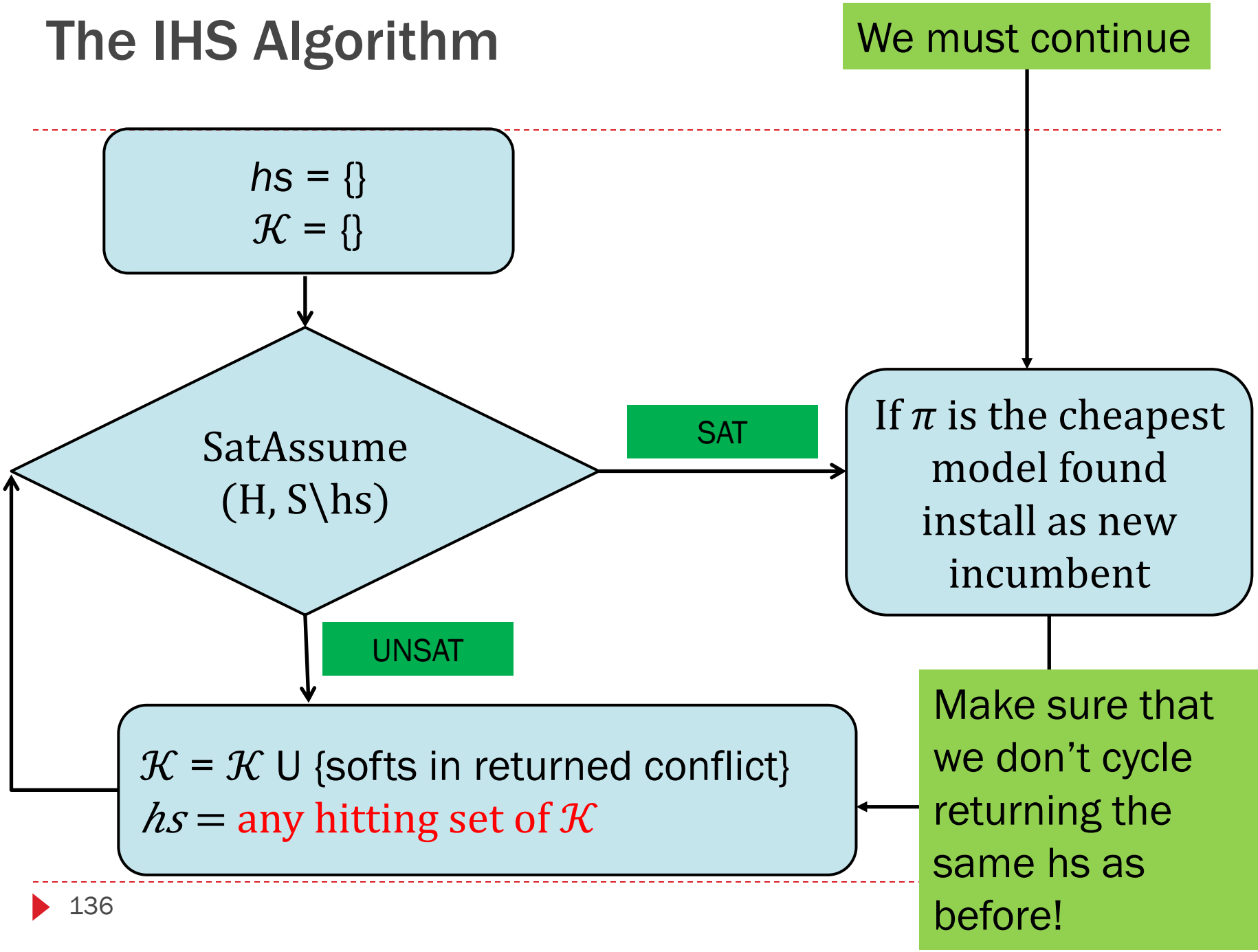
The IHS Algorithm



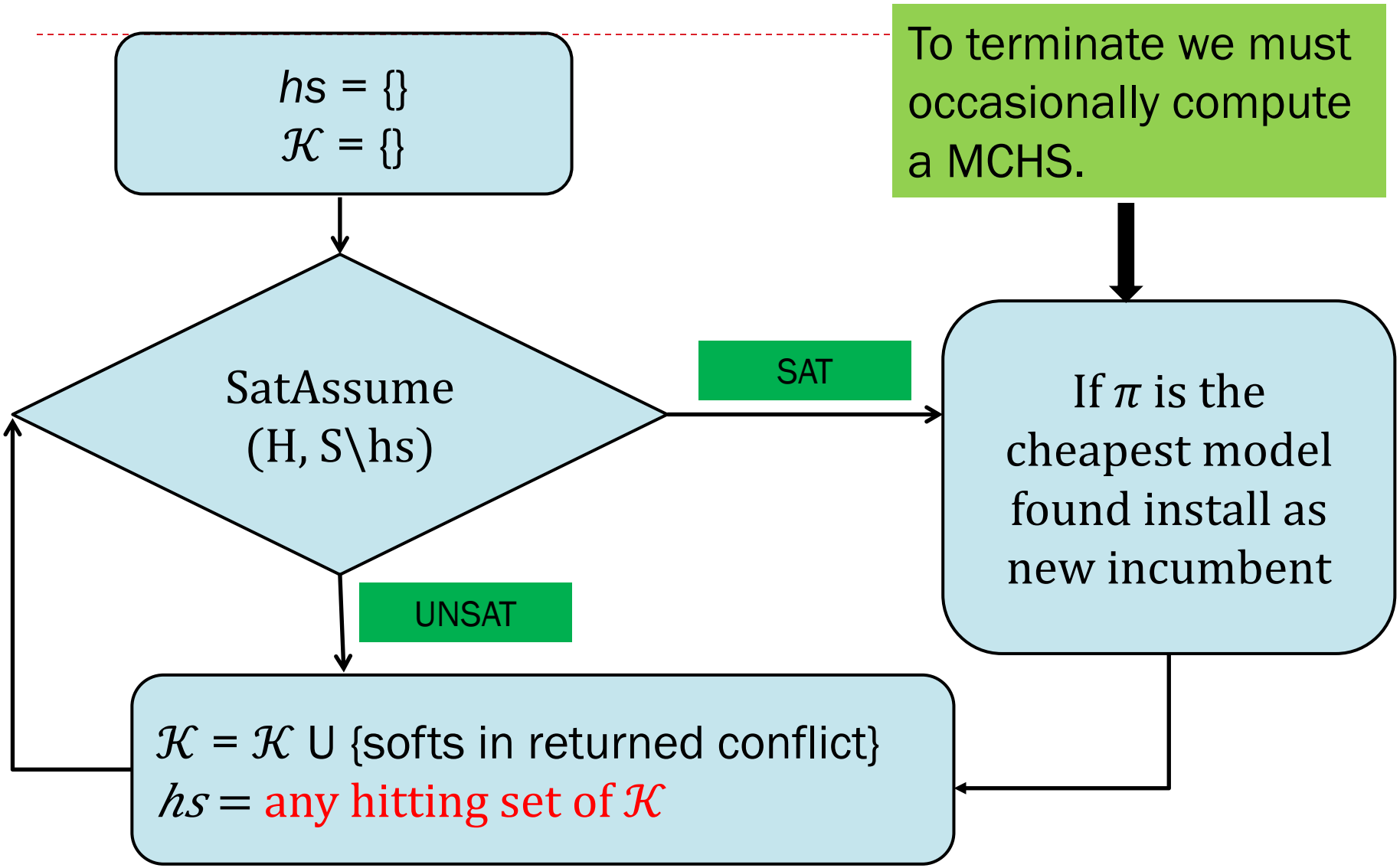
The IHS Algorithm

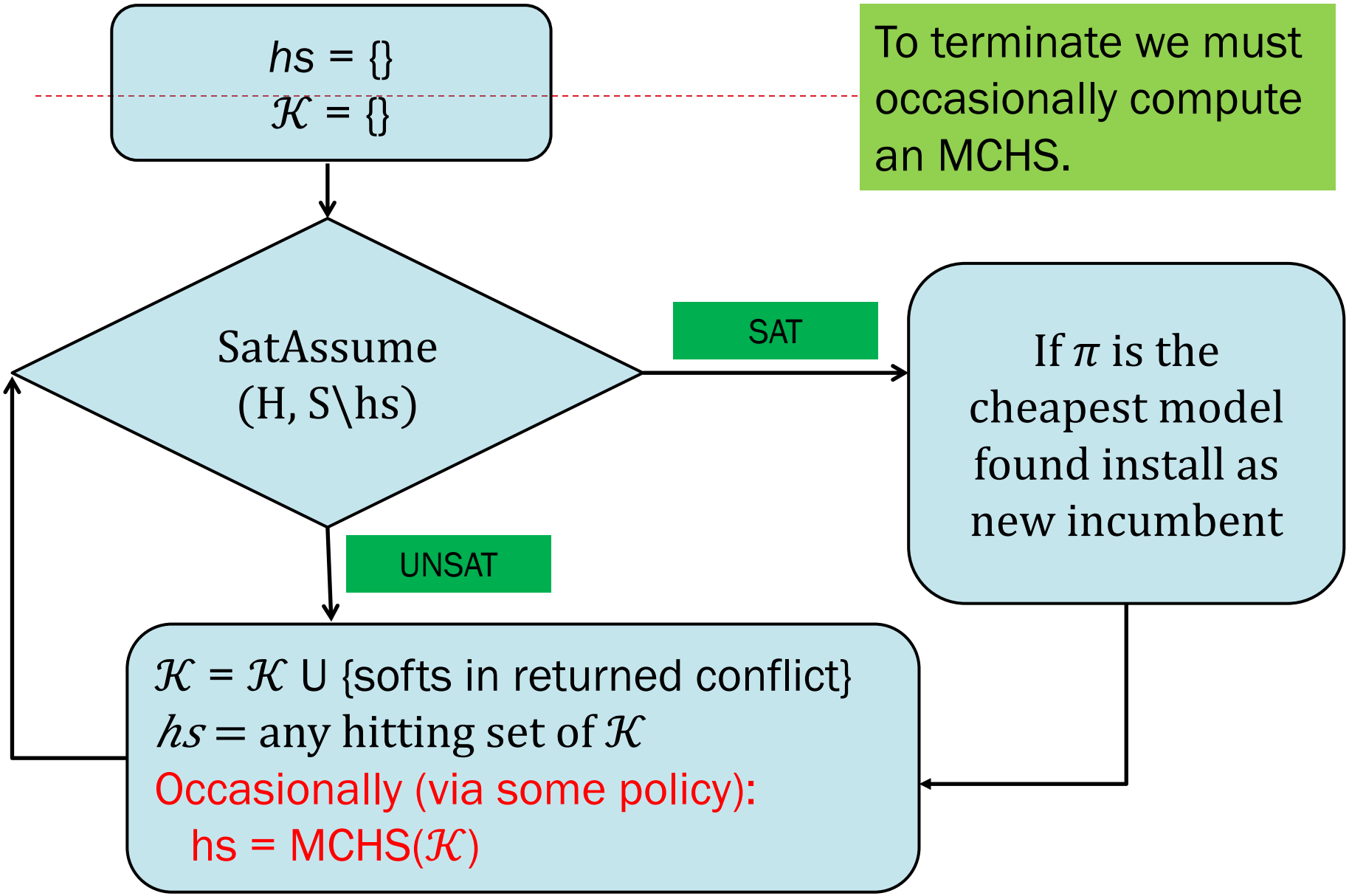


The IHS Algorithm

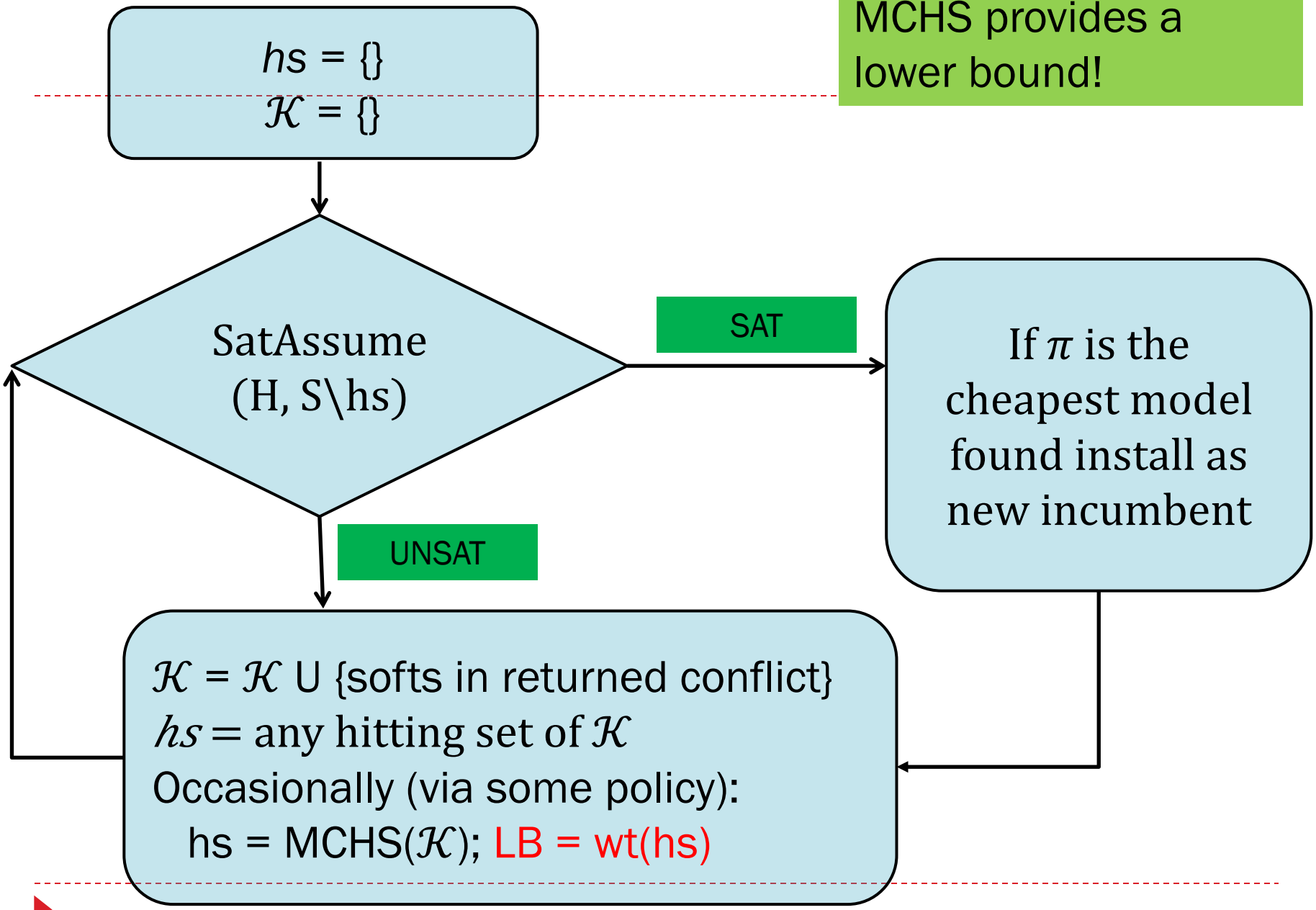


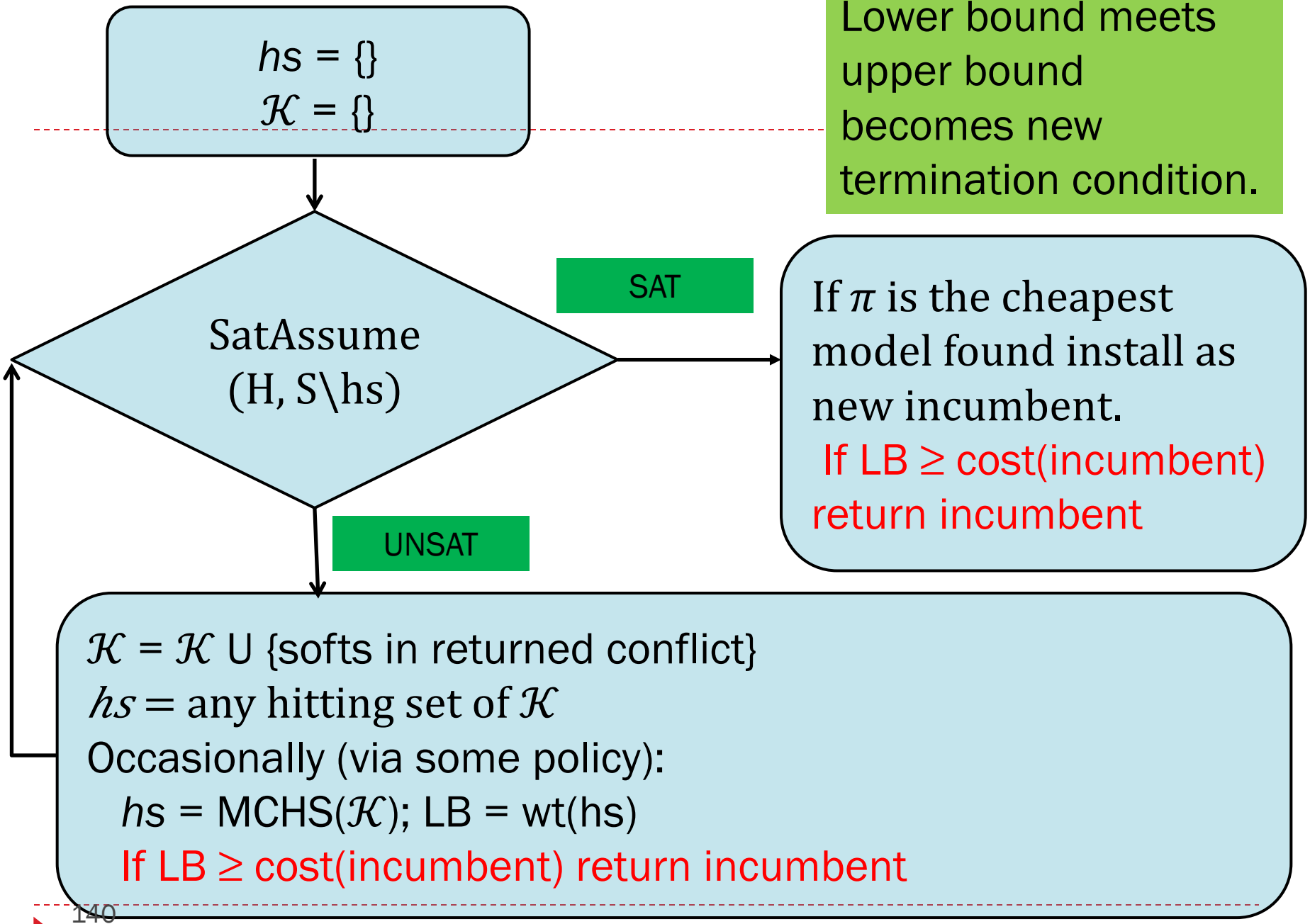
The IHS Algorithm





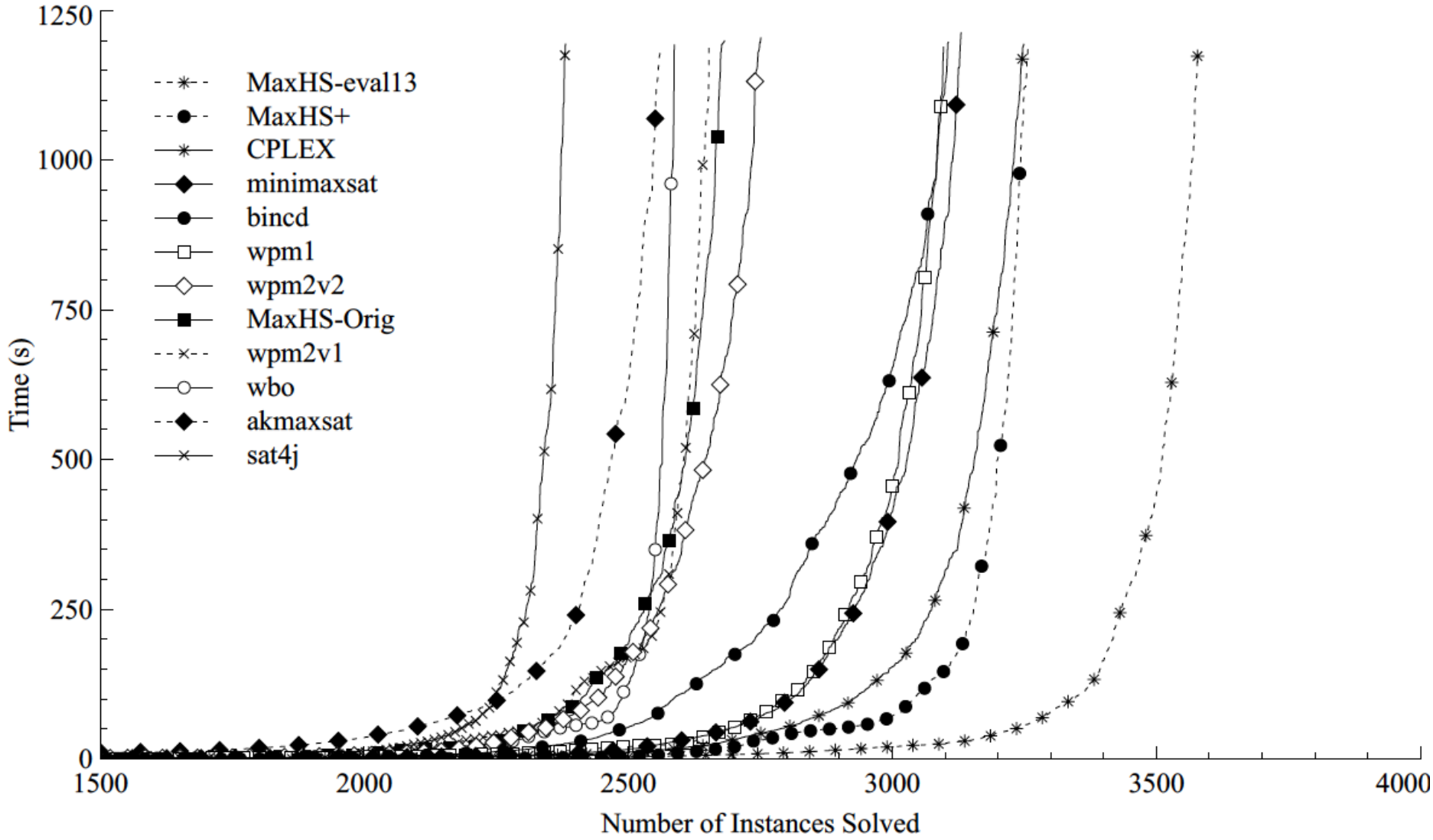
To terminate we must occasionally compute an MCHS.


MCHS provides a lower bound!
hs = {}
𝒦 = {}
SatAssume (H, S\hs)
SAT
If π is the cheapest model found install as new incumbent
UNSAT
𝒦 = 𝒦 U {softs in returned conflict}
hs = any hitting set of 𝒦
Occasionally (via some policy):
hs = MCHS(𝒦); LB = wt(hs)



Lower bound meets upper bound becomes new termination condition.

MaxHS Performance 2013



MaxHS Development

- Numerous improvements since 2013.
- Measuring the number of problems solved within a time limit of 1800 sec. and 3.5 GB (on the same machine) here is how the software has improved.

2013	2014	2015
4410	4613	4862

Recent Developments

- Use CPLEX to compute non-optimal hitting sets
- Use call backs in CPLEX. If CPLEX finds a feasible solution that is better than the current best solution, we stop CPLEX and use its feasible solution as a non-optimal hitting set.

Reduced cost fixing (CP-2017).

- By finding feasible but not optimal solutions using non-optimal hitting sets, we have an upper bound.
- The cost of a MCHS to the current set of cores is a lower bound.
- These two bounds allow us to use the OR technique of reduced cost fixing.

- Fahiem Bacchus, Antti Hyttinen, Matti Jarvisalo, and Paul Saikko; **Reduced Cost Fixing in MaxSAT, CP 2017**

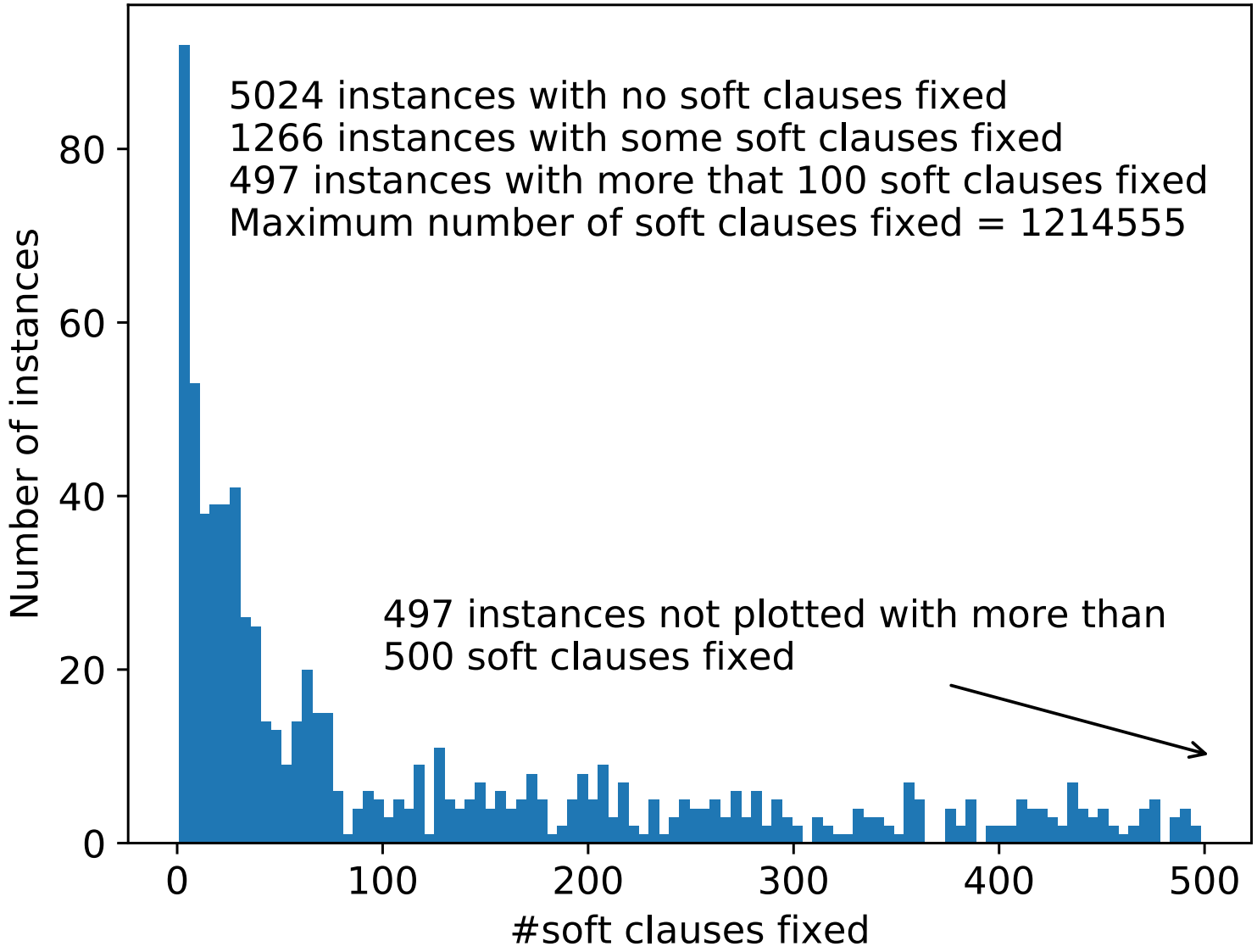
Reduced cost fixing (CP 2017).

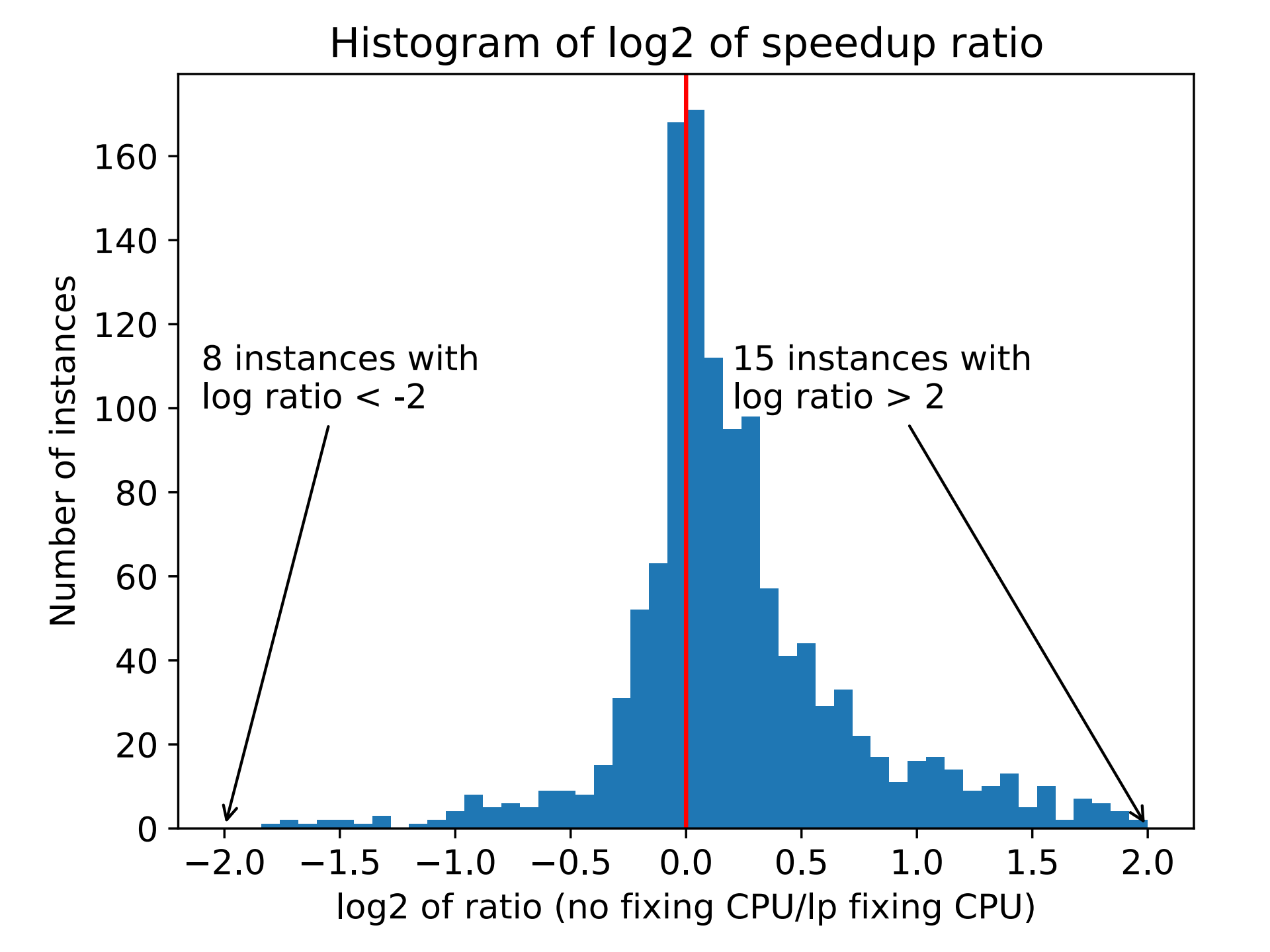
- Solve the Linear program arising from the linear relaxation of the current CPLEX model.
  - The optimal LP solution provides a “derivative” cost for changing the value of the variables that have been set to their upper or lower bound in the optimal solution. These are called the reduced costs of the variables.
  - The LP variables are b-variables set to 0 (satisfy a soft clause) or 1 (falsify a soft clause). So if in the LP the cost of the LP solution + reduced cost(bi) > UPPER BOUND, we can fix that b-variable to 0 converting a soft clause to a hard clause in the SAT model
    - No model with that variable set to 1 will have cost less than the current incumbent.

Reduced cost fixing (CP 2017).

- Similar logic applies to some b-variables set to 1: no better model exists if we require that soft clause to be satisfied—so always falsify it in the SAT model.

Amount of soft clause fixing




Histogram of log2 of speedup ratio
Number of instances
log2 of ratio (no fixing CPU/lp fixing CPU)
8 instances with
log ratio < -2
15 instances with
log ratio > 2
0
20
40
60
80
100
120
140
160
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

MaxSMT

Implicit Hitting Set Algorithms for Maximum Satisfiability Modulo Theories; Katalin Fazekas, Fahiem Bacchus, Armin Biere IJCAR-2018

- Cplex--hitting set optimization
- SAT—propositional reasoning
- SMT—theory reasoning

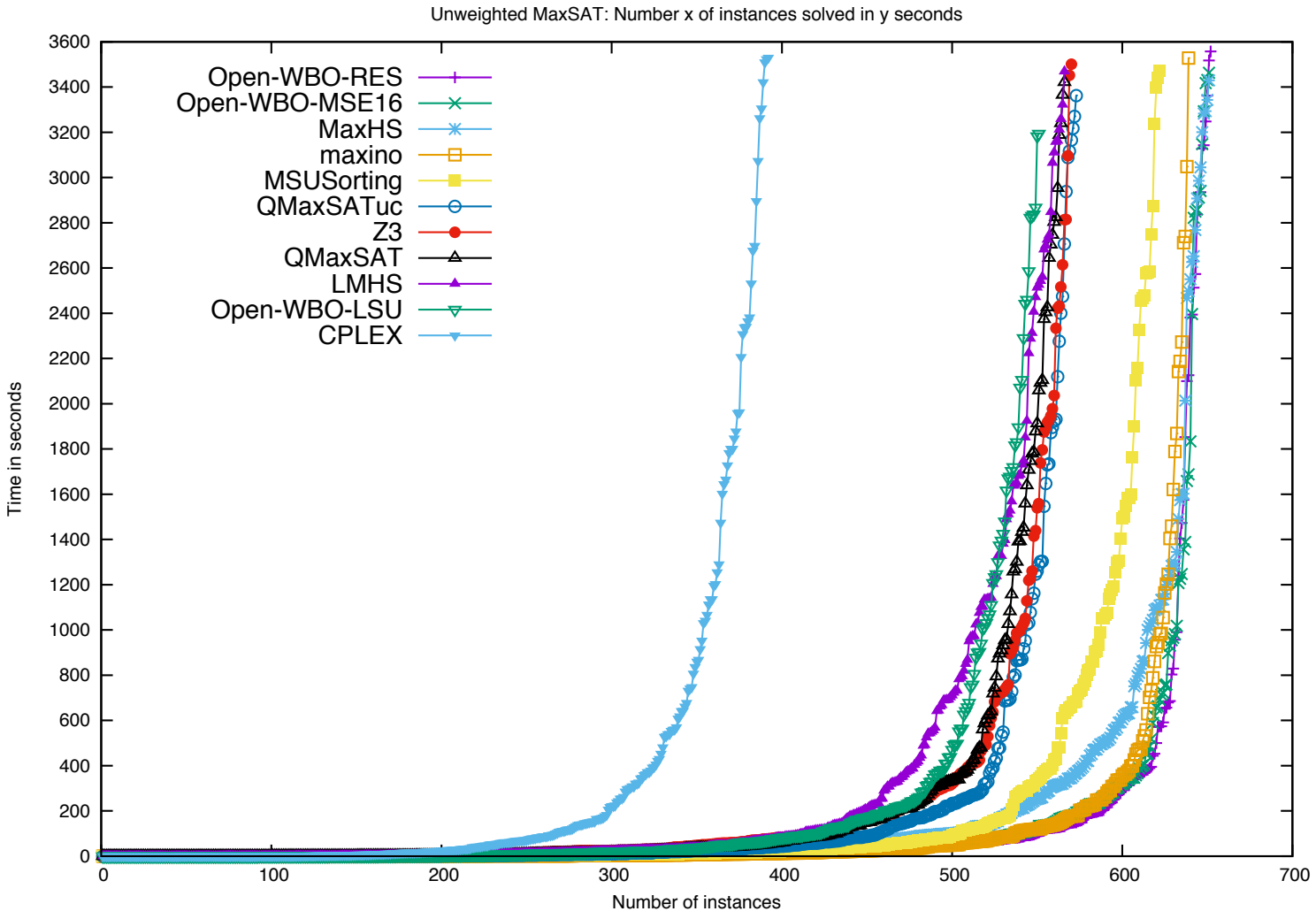
- We provided an abstract reasoning calculus that allows one to mix these types of reasoning in flexible ways so as to solve optimization problems with SMT theories.

MaxHS in the 2017 Evaluation UNWEIGHTED

880 instances

Solver	#Solved	Time (Avg)
Open-WBO-RES	652	129.9
MaxHS	651	182.61
maxino	639	99.14
MSUSorting	622	171.96
QMaxSATuc	573	165.19

MaxHS in the 2017 Evaluation UNWEIGHTED

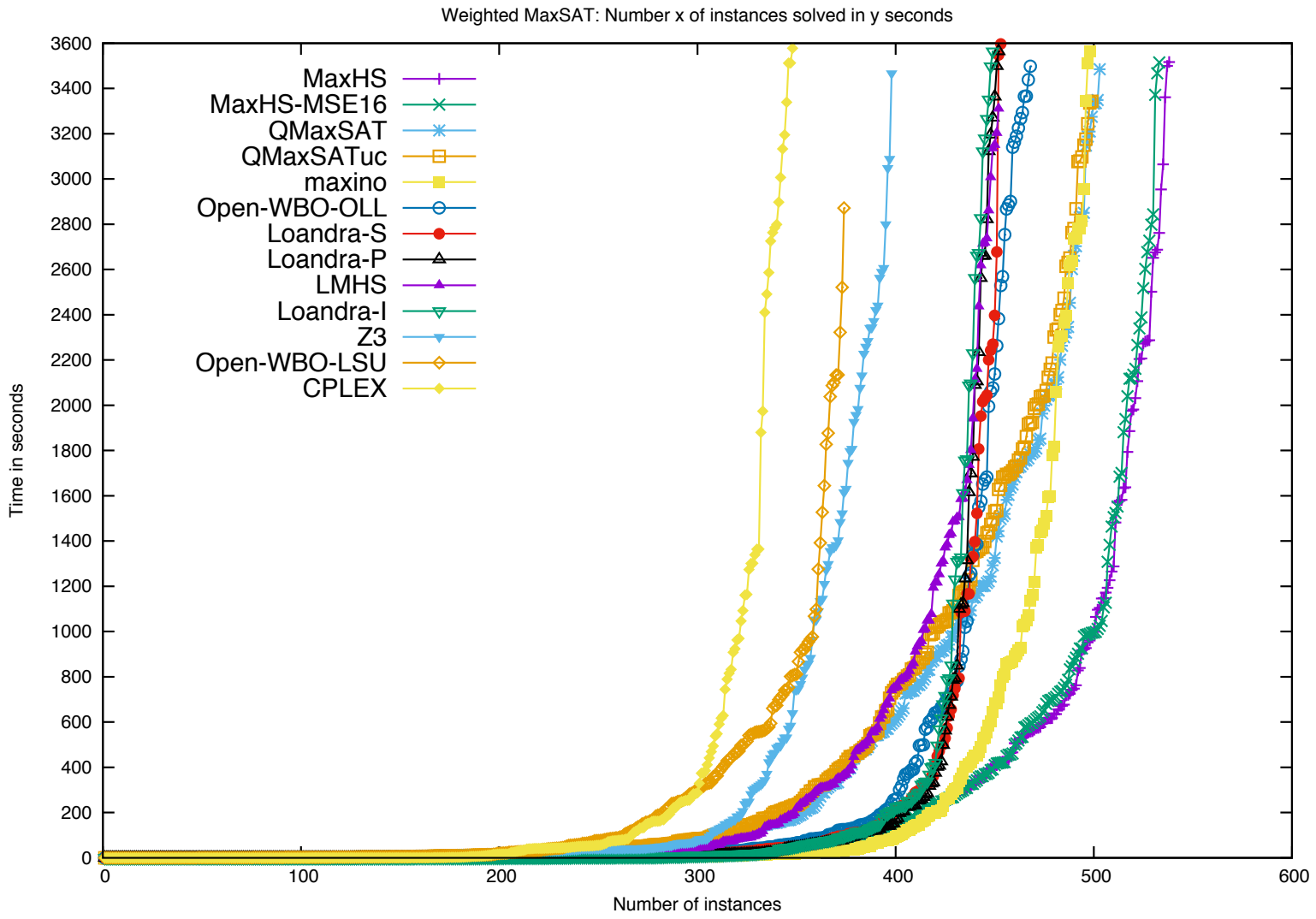


MaxHS in the 2017 Evaluation WEIGHTED

767 instances

Solver	#Solved	Time (Avg)
MaxHS	538	236.46
QMaxSAT	503	385.18
QMaxSATuc	499	397.82
maxino	498	202.1
Open-WBO-OLL	468	231.88

MaxHS in the 2017 Evaluation WEIGHTED



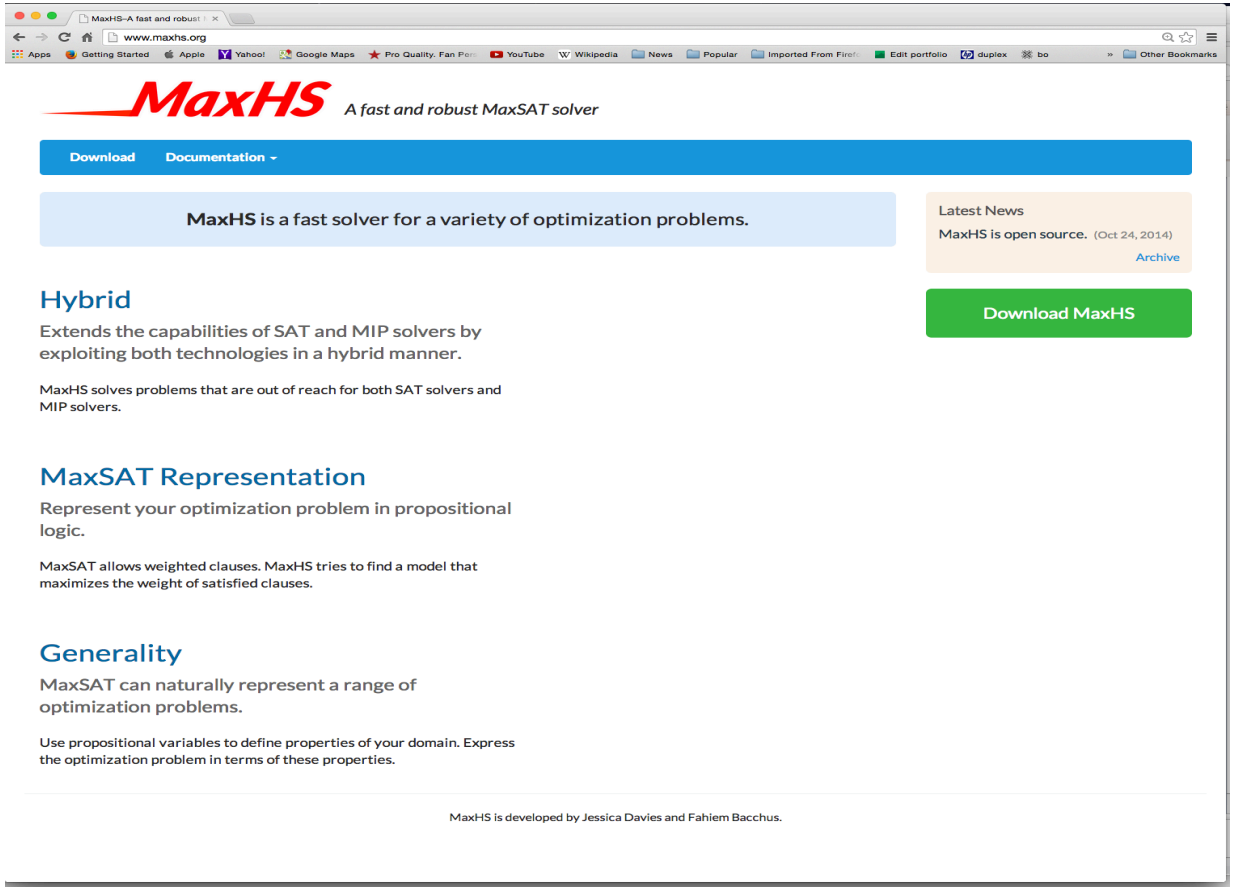
MaxHS in 2018 Evaluation

- Outperformed by RC-2 on the instances used in that evaluation!
- But on larger test set still outperforms RC-2 on weighted instances.

- Nevertheless indicates that the pure SAT based methods do have some effective features.
  - working on importing some key ideas from these solvers into the MaxHS approach.

MaxHS is Open Source

- www.maxhs.org



Future Work

- MaxHS represents only one way of hybridizing IP and SAT. Other methods worth investigating.
  - Could have mixed input models with both clauses and linear constraints.
  - Could use MaxSat as a sub-IP solver for cutting plane generation.
- Applications in matching and other areas.

Thank you

Solving MaxSat with IP

- A simple way to solve MaxSat is to encode it as an IP.
- Add relaxation variables to all soft clauses.
- Convert each clause to a linear constraint.
- Set the objective function to be the minimization of sum of the relaxation variables weighed by their cost.

Solving MaxSat with IP

$$F = \{(\neg l_1, 3), (l_2, 4), (\neg l_3, 1), (l_2 \vee l_3, 10), (l_1 \vee \neg l_2, \infty)\}$$

$$F^b = \{(\neg l_1 \vee b_1), (l_2 \vee b_2), (\neg l_3 \vee b_3), (l_2 \vee l_3 \vee b_4), (l_1 \vee \neg l_2)\}$$

$$\min = 3b_1 + 4b_2 + b_3 + b_4$$

$$1 - l_1 + b_1 \geq 1$$

$$l_2 + b_2 \geq 1 \ldots$$