An Introduction to MAXSAT for Planning Researchers[1]

Fahiem Bacchus

Department of Computer Science
University of Toronto
fbacchus@cs.toronto.edu

[1] Some slides by Jessica Davies and Christian Muise

Outline

1. The MAXSAT Problem.
2. Example uses in planning.
3. Algorithms for solving MAXSAT.
   - 3.1 Brief word on approximation.
   - 3.2 Branch and Bound.
   - 3.3 Formulation as a Mixed Integer Program (MIPS).
   - 3.4 Solving as a sequence of SAT decision problems.
   - 3.5 Hybrid SAT/MIPS approach.
4. Empirical Comparison (as of 2013)

§1. The MAXSAT Problem

- The well-known **SAT** problem is to determine if a boolean formula in Conjunctive Normal Form (CNF) has a satisfying truth assignment
  - A CNF formula is a conjunction of clauses
  - A clause is a disjunction of literals (variables or their negations)
  - $\square$ denotes the empty clause (falsified by every truth assignment)
  - A satisfying assignment assigns *true* to at least one literal of every clause.
- MAXSAT is an **optimization** extension of SAT that asks what is the maximum number of clauses that can be simultaneously satisfied

§1. The MAXSAT Problem

Example

$\mathcal{F} = (\neg x) \wedge (x \vee y) \wedge (\neg y) \wedge (z \vee w)$

§1. The MAXSAT Problem

Example

$\mathcal{F} = (\neg x) \wedge (x \vee y) \wedge (\neg y) \wedge (z \vee w)$
$\mathcal{F}$ is unsatisfiable, so no truth assignment can satisfy all 4 clauses.

§1. The MAXSAT Problem

Example

$\mathcal{F} = (\neg x) \wedge (x \vee y) \wedge (\neg y) \wedge (z \vee w)$
$\mathcal{F}$ is unsatisfiable, so no truth assignment can satisfy all 4 clauses.
Truth assignment $\pi = \{x, \neg y, z, \neg w\}$ satisfies all clauses except $(\neg x)$.

§1. The MAXSAT Problem

Example

$\mathcal{F} = (\neg x) \wedge (x \vee y) \wedge (\neg y) \wedge (z \vee w)$
$\mathcal{F}$ is unsatisfiable, so no truth assignment can satisfy all 4 clauses.
Truth assignment $\pi = \{x, \neg y, z, \neg w\}$ satisfies all clauses except $(\neg x)$.
$\Rightarrow \pi$ is a solution to the MAXSAT problem $\mathcal{F}$.

§1. The MAXSAT Problem

- Some clauses may be more important to satisfy than others
- This can be modeled by associating a positive cost with each clause C that will be incurred if C is *falsified*

Example

$\mathcal{F} = (\neg x, \infty) \wedge (x \vee y, 4) \wedge (\neg y, 1) \wedge (z \vee w, \infty)$

§1. The MAXSAT Problem

- Some clauses may be more important to satisfy than others
- This can be modeled by associating a positive cost with each clause C that will be incurred if C is *falsified*
- If it is mandatory to satisfy C, its cost is ∞ and C is called **hard**
- Otherwise, C is called **soft**

Example

$\mathcal{F} = (\neg x, \infty) \wedge (x \vee y, 4) \wedge (\neg y, 1) \wedge (z \vee w, \infty)$
In $\mathcal{F}$, $(\neg x, \infty)$ is a **hard** clause, and $(x \vee y, 4)$ is **soft** with cost 4.

§1. The MAXSAT Problem

- A truth assignment π has cost equal to the sum of the costs of the clauses it falsifies
- Goal: find an optimal feasible truth assignment, i.e., a truth assignment of minimum **finite** cost $mincost(\mathcal{F})$

Example

$\mathcal{F} = (\neg x, \infty) \wedge (x \vee y, 4) \wedge (\neg y, 1) \wedge (z \vee w, \infty)$
$\pi = \{\neg x, y, z, \neg w\}$ satisfies all clauses except $(\neg y, 1)$
π is optimal: $mincost(\mathcal{F})$=$cost(\pi)$=1.

§1. Notes

- We use $cost(C)$ to denote the cost of clause C.
- A solution must satisfy all hard clauses (else its cost will be infinite).
- A solution also satisfies a **maximum** total cost of soft clauses.
  - Casting as minimization problem more closely corresponds to how most MAXSAT solvers work.
- Many solutions might exist—typically we are only interested in finding one, sometimes only interested in finding out the cost of a solution.

§1.1. Categories of MAXSAT

- MAXSAT (ms) (standard MAXSAT): no hard clauses and all clause have weight 1.
  - Solution maximizes the number of satisfied clauses.
- Weighted MAXSAT (wms): no hard clauses.
- Partial MAXSAT (pms): have hard clauses but all soft clauses have weight 1.
- Weighted Partial MAXSAT (wpms): the version we have defined here (subsumes all other versions).

- Standard MAXSAT is most interesting for theory: it already has sufficient structure for theoretical insights.
- Other versions mostly an artifact of the limitations of earlier MAXSAT solvers.

§1.2. Complexity of MAXSAT

- MAXSAT is complete for the the class FP^{NP}.
  - The class of binary relations $f(x, y)$ where given x we can compute y in polynomial time given access to an NP oracle.
  - Note the oracle can only be called a polynomial number of times since the computation takes poly-time.
  - This complexity class includes the Traveling Salesman.
- Various special cases like 2-MAXSAT (all clauses of length two) which are easy for SAT remain hard for MAXSAT (although 2-MAXSAT can be closely approximated using semidefinite programming methods).

§1.3. Approximating MAXSAT

- There are also limits to how well we can approximate MAXSAT.
  - For standard MAXSAT (no hard clauses, weight one clauses), there is some approximation ratio that can be achieved by a polytime computation, but not every approximation ratio can be achieved in polytime.
  - Formally, MAXSAT is APX-complete (class of NP optimization problems that admit a constant-factor approximation algorithm), but has no PTAS (polynomial-time approximation scheme) unless $NP = P$.
- In practice MAXSAT tends to be much harder than SAT, but this depends on the number and structure of the soft clauses.

§2. Uses of MAXSAT in Planning

- Many optimization problems can be naturally encoded in MAXSAT
  - Traveling Salesman
  - MaxCut, MaxClique
  - Most probable explanation problems (MPE)
- We look at some uses in planning
  - Computing optimal ordering relaxations of a sequential plan.
  - Computing optimal plans under different criteria for optimality.
  - Computing A^* heuristics for optimal planning.

§2.1. Optimal relaxations of a sequential plan

"Optimally Relaxing Partial-Order Plans with MAXSAT", Muise, McIlraith and Beck (ICAPS-12)

- The best planners compute sequential plans. But partially ordered plans (POP) are more flexible as execution time.
- Take the found sequential plan $\mathcal{A} = \langle a_1, a_2, \ldots, a_n \rangle$ and create a POP by
  - removing as many ordering constraints as possible
  - removing as many actions as possible
  - while preserving plan correctness
- Augment $\mathcal{A}$ with initial state action a_I and goal action a_G. a_I adds initial state facts, a_G has goals as precondition.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- No self loops.
- Include a_I and a_G.
- If an ordering is used, include the actions.
- If we include an action, order it after (before) a_I (a_G).
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- No self loops.
- Include a_I and a_G.
- If an ordering is used, include the actions.
- If we include an action, order it after (before) a_I (a_G).
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ No self loops.
- Include a_I and a_G.
- If an ordering is used, include the actions.
- If we include an action, order it after (before) a_I (a_G).
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ No self loops.
- Include a_I and a_G.
- If an ordering is used, include the actions.
- If we include an action, order it after (before) a_I (a_G).
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ — No self loops.
- $(x_I) \wedge (x_G)$ — Include a_I and a_G.
- If an ordering is used, include the actions.
- If we include an action, order it after (before) a_I (a_G).
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ — No self loops.
- $(x_I) \wedge (x_G)$ — Include a_I and a_G.
- If an ordering is used, include the actions.
- If we include an action, order it after (before) a_I (a_G).
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ — No self loops.
- $(x_I) \wedge (x_G)$ — Include a_I and a_G.
- $\kappa(a_i, a_j) \rightarrow x_i \wedge x_j$ — Ordering implies actions.
- If we include an action, order it after (before) a_I (a_G).
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ — No self loops.
- $(x_I) \wedge (x_G)$ — Include a_I and a_G.
- $\kappa(a_i, a_j) \rightarrow x_i \wedge x_j$ — Ordering implies actions.
- If we include an action, order it after (before) a_I (a_G).
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ — No self loops.
- $(x_I) \wedge (x_G)$ — Include a_I and a_G.
- $\kappa(a_i, a_j) \rightarrow x_i \wedge x_j$ — Ordering implies actions.
- $x_i \rightarrow \kappa(a_I, a_i) \wedge \kappa(a_i, a_G)$ — Order actions with a_I and a_G.
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ — No self loops.
- $(x_I) \wedge (x_G)$ — Include a_I and a_G.
- $\kappa(a_i, a_j) \rightarrow x_i \wedge x_j$ — Ordering implies actions.
- $x_i \rightarrow \kappa(a_I, a_i) \wedge \kappa(a_i, a_G)$ — Order actions with a_I and a_G.
- Enforce the transitive closure.

§2.1. Optimal relaxations

Action Variables and Ordering Variables

x_i	iff a_i in final POP.
$\kappa(a_i, a_j)$	iff $a_i < a_j$ in final POP.
$\Upsilon(a_i, a_j, p)$	True iff a_i supports a_j with fluent p

Basic Clauses

- $\neg\kappa(a, a)$ — No self loops.
- $(x_I) \wedge (x_G)$ — Include a_I and a_G.
- $\kappa(a_i, a_j) \rightarrow x_i \wedge x_j$ — Ordering implies actions.
- $x_i \rightarrow \kappa(a_I, a_i) \wedge \kappa(a_i, a_G)$ — Order actions with a_I and a_G.
- $\kappa(a_i, a_j) \wedge \kappa(a_j, a_k) \rightarrow \kappa(a_i, a_k)$ — Transitive closure.

§2.1. Optimal relaxations

POP Viability Clauses

- If we include action a_j, then every precondition p of a_j must be satisfied by at least one achiever a_i.

- If a_i achieves precondition p for action a_j, then no deleter of p will be allowed to occur between a_i and a_j.

§2.1. Optimal relaxations

POP Viability Clauses

- If we include action a_j, then every precondition p of a_j must be satisfied by at least one achiever a_i.

$$x_j \rightarrow \bigwedge_{p \in PRE(a_j)} \bigvee_{a_i \in \textbf{adders}(p)} \left[\kappa(a_i, a_j) \wedge \Upsilon(a_i, a_j, p) \right]$$

- If a_i achieves precondition p for action a_j, then no deleter of p will be allowed to occur between a_i and a_j.

§2.1. Optimal relaxations

POP Viability Clauses

- If we include action a_j, then every precondition p of a_j must be satisfied by at least one achiever a_i.

$$x_j \rightarrow \bigwedge_{p \in PRE(a_j)} \bigvee_{a_i \in \textbf{adders}(p)} \left[\kappa(a_i, a_j) \wedge \Upsilon(a_i, a_j, p) \right]$$

- If a_i achieves precondition p for action a_j, then no deleter of p will be allowed to occur between a_i and a_j.

$$\Upsilon(a_i, a_j, p) \rightarrow \left[\bigwedge_{a_k \in \textbf{deleters}(p)} x_k \rightarrow \kappa(a_k, a_i) \vee \kappa(a_j, a_k) \right]$$

§2.1. Optimal relaxations

Soft Clauses

1. $wt(\neg\kappa(a_i, a_j)) = 1, \quad \forall a_i, a_j \in \mathcal{A}$
2. $wt(\neg x_i) = 1 + |\mathcal{A}|^2, \quad \forall a \in \mathcal{A} \setminus \{a_I, a_G\}$

Meaning

- From #1 truth assignments that impose fewer ordering constraints are preferred—every true $\kappa(a_i, a_j)$ incurs a cost.
- From #2 truth assignments that include fewer actions are preferred—every true x_i incurs a cost.
- There are at most $O(n^2)$ ordering facts $\kappa(a_i, a_j)$, so we would prefer to remove a single action even at the cost of adding all ordering constraints.

§2.1. Optimal relaxations

The authors report pretty good empirical performance, showing that MAXSAT is competitive (and very natural) for this task.

The MAXSAT solver used by the authors is now quite out of date, so better performance could now be expected.

§2.2. Computing Optimal Plans.

- MAXSAT can be used to compute optimal plans under various optimality criteria (but not all criteria are easy to achieve).
- Start with a standard SAT encoding of the planning problem.

§2.2. Computing Optimal Plans.

- Each state layer *i* has time indexed fluent propositions—p_i is true when fluent *p* is true at time step *i* of the plan.
- Each action layer has time indexed action propositions—a_i is true action when *a* is executed at time step *i* of the plan.

§2.2. Computing Optimal Plans.

Various SAT clauses are added to relate these propositions.

- Fluents true at step *i* correspond to the effect and non-effects of the actions executed at step *i*.
- Actions at step *i* are only executed if their preconditions hold at state layer $i - 1$
- The goal fluents are true at the final state layer.
- Different conditions $\mathcal{C}$ on which actions can be executed together at step *i*.

§2.2. Computing Optimal Plans.

- Typically SAT-planners grow the number of steps of this structure until the formula is satisfiable—the setting of the action variables in the satisfying model is the plan.
- By incrementing the number of steps SAT-planners can find a “step” optimal plan.
- Unfortunately, this might not mean much.

§2.2. Computing Optimal Plans.

- If we restrict the encoding to allow only one action per step, then the optimal step plan will minimize the number of actions.
  - This does not help much if actions have varying costs.
  - Other problem is that the size of the SAT encoding grows with the number of layers. The restriction to sequential plans means many more layers.
- Smaller encodings are derived by techniques to maximize the number of allowed actions per step (parallel actions)
  - Minimum number of parallel steps does not mean minimizing number of actions.
  - If actions have different durations it does not minimize makespan either.

§2.2. Computing Optimal Plans.

- Add to the SAT encoding the soft clauses

$$\{(\neg a_i) \mid a \in \text{Actions} \wedge i \in [1, \ldots n]\}$$

with

$$cost(`(\neg a_i)`) = cost(a)$$

- Let $\mathcal{C}$ be the condition on simultaneous actions at each step. $\mathcal{C}$ might be "allow zero or one", "allow zero or more graphplan non-conflicting actions" etc.
- The soft clauses give a MAXSAT problem that computes a plan with minimal action cost **among those plans** that have n steps of type $\mathcal{C}$.
- The true optimal (lowest cost) plan might require $n+1$ $\mathcal{C}$-steps!

§2.2. Computing Optimal Plans.

Options:

- Find a number of $\mathcal{C}$-steps n that you can prove is larger than the number $\mathcal{C}$-steps of any optimal plan, use this value of n and solve the MAXSAT problem.
  - E.g., if $\mathcal{C}$ is "zero or one" (sequential plans), then the cost of any satisfying plan divided by the cost of the minimum cost action provides such a bound n.
  - With highly varied action costs this bound can be very large–yields a very large MAXSAT problem that might be impossible to solve.
- Be satisfied by $\mathcal{C}$-step optimal plans for a fixed n.

§2.2. Computing Optimal Plans.

Another approach was suggested in "Partial Weighted MaxSAT for Optimal Planning", Robinson, Gretton, Pham, and Sattar (PRICAI-10)

- Their idea is to have two MAXSAT encodings:
  1. *n* graphplan parallel steps.
  2. #1 along with an arbitrary step delete relaxed suffix.
- If both encodings produce an identical cost plan encoding #1 gives an true cost optimal plan.

§2.2. Computing Optimal Plans.

- Intuitively for fixed n encoding #1 provides an upper bound on the optimal cost plan, while encoding #2 provides a lower bound (it is cheaper due to the delete relaxed phase).
- However, the #2 might give a weak lower bound by pushing actions with expensive deletes into the delete relaxed phase.

§2.3. Computing A^* heuristics

- h^+ the cost of an optimal delete relaxed plan is known to be a very good admissible heuristic for A^*.
- But computing h^+ is hard.
- We can compute h^+ using MAXSAT.
  - A simple way is to build a layered SAT encoding ignoring delete effects, then solve this optimally as described above.
  - This can yield a large MAXSAT problem that is expensive to solve–not conducive for use inside of an A^* search.
  - Might work well for delete-free optimal planning.

§2.3. Computing A^* heuristics

"MAXSAT Heuristics for Cost Optimal Planning", Zhang, and Bacchus (AAAI-2012) suggests an approximate MAXSAT encoding.

- Ordering constraints between the relaxed actions are ignored, this yields a much smaller MAXSAT encoding.
- The optimal solution is a lower bound on h^+.
- A constraint generation approach can be used to incrementally account for the ordering constraints. After a finite number of additional constraints are added the encoding will compute h^+ exactly.
- Works fairly well, but LM-Cut already gets very close to h^+, so the real value of this approach probably lies in computing heuristics more powerful than h^+.

§2.4. Optimal planning with MAXSAT
Possible ways forward

- Find better ways to bound the number of $\mathcal{C}$-steps needed to admit an optimal plan so as to obtain smaller and easier to solve MAXSAT encodings.
- Find ways to compute improved heuristics and deadend detection using MAXSAT and SAT to augment A^*.
- This would lever the constant improvement in MAXSAT solvers.

§3. Algorithms for Solving MAXSAT

- The main focus will be on Exact Algorithms.
- First a few words about Approximations.

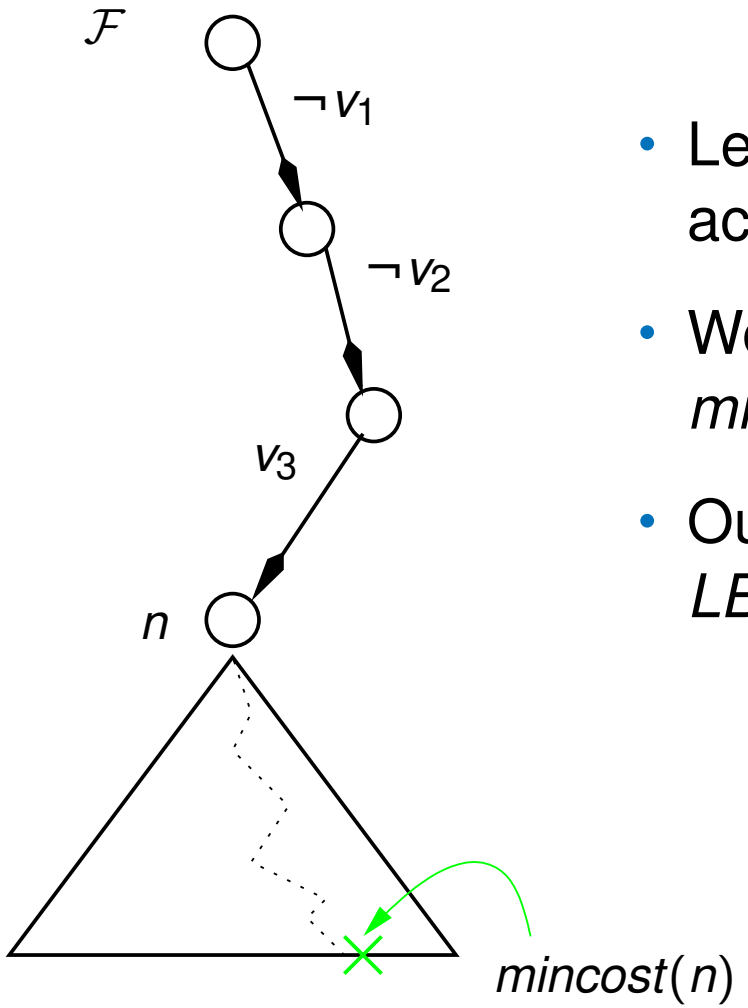
§3.1. Approximating MAXSAT

- Important theoretical work has been done on approximation. Focused on standard MAXSAT (unit weights and no hard clauses). For 2-MAXSAT very good approximations can be produced using semi-definite programming relaxations.
- Modifications of SAT local search methods have been used. Not much recent work.
- Often the hard clauses for MAXSAT are quite structured, and local search methods have a difficult time satisfying the hard clauses producing solutions of infinite cost.
- Linear relaxations might work but have not been much used.

§3.2. Branch and Bound

- Backtracking search through the space of partial truth assignments
- Find initial upper bound (UB) on $\mathit{mincost}(\mathcal{F})$ (local search or some satisfying assignment to the hard clauses)
- Update the UB when a better complete assignment is found
- Relies on a lower bound function to prune the search space

§3.2. Branch and Bound



- Let $mincost(n)$ be the minimum achievable cost under node n
- We can backtrack from n only if we know $mincost(n) \geq UB$
- Our goal: calculate a lower bound s.t. $LB \leq mincost(n)$ and $LB \geq UB$

§3.2. Lower Bounds

Example

$\mathcal{F} = \ldots \wedge (x, 2) \ldots \wedge (\neg x, 3) \ldots$

§3.2. Lower Bounds

Example

$\mathcal{F} = \ldots \wedge (x, 2) \ldots \wedge (\neg x, 3) \ldots$
Ignoring clause costs, $\kappa = \{(x, 2) \wedge (\neg x, 3)\}$ is inconsistent.

§3.2. Lower Bounds

Example

$\mathcal{F} = \ldots \wedge (x, 2) \ldots \wedge (\neg x, 3) \ldots$
Ignoring clause costs, $\kappa = \{(x, 2) \wedge (\neg x, 3)\}$ is inconsistent.

Let $\kappa' = \{(\square, 2) \wedge (\neg x, 1)\}$.

§3.2. Lower Bounds

Example

$\mathcal{F} = \ldots \wedge (x, 2) \ldots \wedge (\neg x, 3) \ldots$
Ignoring clause costs, $\kappa = \{(x, 2) \wedge (\neg x, 3)\}$ is inconsistent.

Let $\kappa' = \{(\square, 2) \wedge (\neg x, 1)\}$.
Then κ' is MAXSAT-equivalent to κ: the cost of each truth assignment is preserved.

§3.2. Lower Bounds

Example

$\mathcal{F} = \ldots \wedge (x, 2) \ldots \wedge (\neg x, 3) \ldots$
Ignoring clause costs, $\kappa = \{(x, 2) \wedge (\neg x, 3)\}$ is inconsistent.

Let $\kappa' = \{(\square, 2) \wedge (\neg x, 1)\}$.
Then κ' is MAXSAT-equivalent to κ: the cost of each truth assignment is preserved.

Let $\mathcal{F}' = \mathcal{F} - \kappa + \kappa'$.

§3.2. Lower Bounds

Example

$\mathcal{F} = \ldots \wedge (x, 2) \ldots \wedge (\neg x, 3) \ldots$
Ignoring clause costs, $\kappa = \{(x, 2) \wedge (\neg x, 3)\}$ is inconsistent.

Let $\kappa' = \{(\square, 2) \wedge (\neg x, 1)\}$.
Then κ' is MAXSAT-equivalent to κ: the cost of each truth assignment is preserved.

Let $\mathcal{F}' = \mathcal{F} - \kappa + \kappa'$.
Then $\mathcal{F}'$ is MAXSAT-equivalent to $\mathcal{F}$, and the cost of $\square$ has been incremented by 2—a lower bound.

§3.2. Lower Bounds

1. Detect an inconsistent subset κ of the current formula
   - e.g. $\kappa = \{(x, 2) \wedge (\neg x, 3)\}$
2. Apply sound transformation rules to the clauses in κ that result in an increment to the cost of the empty clause $\square$
   - e.g. κ replaced by $\kappa' = \{(\square, 2) \wedge (\neg x, 1)\}$
   - The cost of $\square$ is a lower bound
3. Repeat 1 and 2 until no further increment to the LB is possible

§3.2. Detecting Inconsistent Subformulas

Treat the soft clauses as if they were hard and then:

- use **Unit Propagation** (UP) to efficiently detect κ: find and instantiate literals appearing in unit clauses until an empty clause is generated
- If no unit clauses exist, use **Failed Literal Detection** (FLD)
- UP and FLD are implemented using the *watched literals* data structure
- Finding inconsistent subformulas is very fast

§3.2. Transforming the Formula

- Various patterns of formula sets that can be transformed have been identified.
- The transformations are mostly (all?) instances of sequences of applications of the MAXRES rule
- MAXRES is a sound and complete inference rule for MAXSAT [“Resolution for Max-Sat” by Bonet, Levi, and Manya (Artificial Intelligence 2007]

§3.2. MAXRES

- MAXRES is a rule of inference that like ordinary resolution takes as input two clauses and produces new clauses.
- Unlike resolution MAXRES (a) removes the input clauses and (b) produces multiple new clauses.

§3.2. MAXRES

MAXRES $[(x, a_1, \ldots, a_s, w_1), (\neg x, b_1, \ldots, b_t, w_2)] =$

$(a_1, \ldots, a_s, b_1, \ldots, b_t, \min(w_1, w_2))$	Regular Resolvant
$(x, a_1, \ldots, a_s, w_1 - \min(w_1, w_2))$	Cost Reduced Input
$(\neg x, b_1, \ldots, b_t, w_2 - \min(w_1, w_2))$	One will vanish
$(x, a_1, \ldots, a_s, \neg b_1, \min(w_1, w_2))$	Compensation Clauses
$(x, a_1, \ldots, a_s, b_1, \neg b_2, \min(w_1, w_2))$	$\ldots$
$(x, a_1, \ldots, a_s, b_1, \ldots, b_{t-1}, \neg b_t, \min(w_1, w_2))$	$\ldots$
$(\neg x, b_1, \ldots, b_t, \neg a_1, \min(w_1, w_2))$	$\ldots$
$(\neg x, b_1, \ldots, b_t, a_1, \neg a_2, \min(w_1, w_2))$	$\ldots$
$(\neg x, b_1, \ldots, b_t, a_1, \ldots, a_{s-1}, \neg a_s, \min(w_1, w_2))$	$\ldots$

§3.2. MAXRES

- MAXRES preserves the cost of every truth assignment—not difficult to prove—let π be an arbitrary truth assignment and consider the three cases
  1. π falsifies only $(x, a_1, \ldots, a_s)$
  2. π falsifies only $(\neg x, b_1, \ldots, b_t)$
  3. π falsifies neither inputs $(\neg x, b_1, \ldots, b_t)$.
- Bonet et al. also give a systematic procedure where by MAXRES can derive the empty clause $(\square, \mathit{Opt})$ with weight Opt equal to the optimal cost.

§3.2. Lower Bounds using MAXRES

- Advantages
  - Transformations allow the lower bound to be incremental as search moves down the branch to descendent nodes
- Disadvantages
  - Transformations may not pay off: the size and structure of the formula can be adversely effected
  - Therefore in practice transformations are restricted to simple cases on short clauses
  - Many easily detected inconsistencies will not be captured by any simple transformation
  - All work is lost upon backtrack
- Solvers that use such lower bounds include MiniMaxSat [Heras et al. 2008], MaxSatz$_c$ [Li et al. 2010].

§3.2. Lower Bounds using Hitting Set

FC	*ic*
$\square$	$\{(x, 2), (\neg x, 4)\}$
(t)	$\{(\neg x, 4), (\neg y, x, 5), (y, t, 10)\}$
$(x, \neg z)$	$\{(y, \neg z, 1), (\neg y, x, 5)\}$

$mincost(n) \geq 4 + 1 = 5$

- FC is a set of (possibly **learnt**) clauses falsified at the current node n
- $ic(C)$ is a set of soft clauses in the input formula such that C can be derived from $ic(C) \wedge hard(\mathcal{F})$
- The cost of the Min Hitting Set of $\{ic(C) : C \in FC\}$ is an LB [Davies et al. 2010]

§3.2. Lower Bounds using Hitting Set

- The learnt clauses in *FC* can be derived using techniques from SAT solving
- Solving the Min Hitting Set problem at each node is expensive: use heuristics or linear relaxation to lower bound the optimum cost
- Advantages
  - Captures *all* inconsistencies detected via UP or FLD

§3.3. Solving with a MIPs Solver

- Optimization problems have been studied for decades in the field of operations research (OR).
- In OR the most commonly used tool for solving optimization problems are state-of-the-art Mixed Integer Program Solvers, like IBM's CPLEX.
- These solvers solve problems with linear constraints and objective function where some variables are integers.
- CPLEX is an extremely effective and successful solver, so it is natural to consider using this tool for MAXSAT as well.

§3.3. Solving with a MIPs Solver

- To every soft clause C_i add a new "blocking" variable b_i.

$$(x, \neg y, z, \neg w) \Rightarrow (x, \neg y, z, \neg w, b_1)$$

- Convert every augmented clause into a linear constraint:

$$x + (1 - y) + z + (1 - w) + b_i \geq 1$$

- Each variable is integer in the range $[0 - 1]$.
- Finally add the objective function

$$\text{minimize} \sum_i b_i * cost(C_i)$$

§3.3. Solving with a MIPs Solver

- MIPs solvers use Branch and Cut to solve.
  - Compute a series of linear relaxations and cuts (new linear constraints that cut off non-integral solutions).
  - Sometimes branch on a bound for an integer variable.
- CPLEX uses lots of other techniques, and it is available for free use to academics.
- For standard optimization problems, like minimum hitting sets (set cover) it is extremely effective.
- As we will see it is quite effective on MAXSAT as well.

§3.4. Solving MAXSAT by a Sequence of SAT Instances

- Each SAT instance in the sequence **encodes** a MAXSAT Decision Problem “Is there a truth assignment of cost at most k?”, then k is varied
- Modern algorithms use information obtained from the SAT solver at each stage for the next stage.

§3.4. Sequence of SAT Instances

Simplest version (Een & Sorensson, 2006, MiniSat+). Works only for unit weights:

1. Input MAXSAT CNF Φ
2. Add blocking variable b_i to every soft clause $C_i \in \Phi$
3. Set $k = 0$.
4. If SAT$(\Phi \cup \mathit{CNF}(\sum b_i = k))$ return k
5. Else $k = k + 1$ and repeat.

§3.4. Sequence of SAT Instances

- The function *CNF* converts a linear constraint over propositional variables into a set of **hard** clauses.
- By setting b_i *true* we "block" or remove soft clause C_i—it longer constrains the problem.
- The theory tests if we can satisfy all of the remaining clauses by removing any set of k soft clauses.

§3.4. Sequence of SAT Instances

- We can also impose $\sum b_i < k$, starting with large k and decreasing until we transition from SAT to UNSAT.
- sat4j (Le Berre 2006) uses the large to small k approach. But every time the formula is SAT we can reduce k to be the actual number of soft clauses falsified (must be less than k).
- Binary search on k can also be used.

Pseudo Boolean Constraints

- General linear constraints on Boolean variables are called **pseudo boolean constraints**.
- There are many techniques for converting pseudo boolean constraints into CNF: using adding circuits, sorting networks, BDDs.
- The minisat+ solver implements a number of conversion methods (and its source code is available).
  `minisat.se`.
- Carsten Sinz also has a implementation of his technique available.
  `http://www.carstensinz.de/software.html`

§3.4. Sequence of SAT Instances

The simple version does not use any information returned by the SAT solver. SAT solvers can extract **Unsat Cores**.

- An Unsat Core $\mathcal{C}$ is a subset the clauses that is unsatisfiable: at least one clause from $\mathcal{C}$ must be falsified by any truth assignment
- Hard clauses must be satisfied so can remove them from the core.
- We can also minimize the unsat core using additional calls to the SAT solver to obtain a Minimal Unsatisfiable Set (MUS).
- This leads to the Fu & Malik algorithm (SAT 2005).

§3.4. Fu & Malik

1. Input MAXSAT CNF Φ
2. $k = 0$
3. While *true*
   - 3.1 $(\kappa, SAT?) = \text{SAT}(\Phi)$
   - 3.2 If $SAT?$ return k.
   - 3.3 Else $\Phi =$
     - 3.3.1 Add **new** blocking variable to every $C \in \kappa$.
     - 3.3.2 Add $CNF(\sum_{\text{new } b\text{-variables}} b = 1)$

- We know that at least one soft clause in κ must be falsified: this gives the initial linear constraint. This constraint is over fewer variables than the naive approach.
- At each subsequent step κ is generated **even when the previous cores are blocked**

§3.4. Fu & Malik

- Multiple blocking variables can accumulate in a single soft clause—one is added every time the clause appears in a new core.
- This leads to redundant ways of blocking the same clause and an explosion in the search space.
- Also can’t deal with weighted clauses.

§3.4. Weighted Clauses

- Things get more complex with weighted clauses. We briefly describe two alternative algorithms WPM1 and briefly WPM2.
- These are described more detail in "SAT-based MAXSAT Algorithms", Ansótegui, Bonet, Levy (AIJ 2013)

§3.4. WPM1

1. Input MAXSAT CNF Φ
2. $mincost = 0$
3. While *true*
   - 3.1 $(\kappa, SAT?) = \text{SAT}(\Phi)$
   - 3.2 If $SAT?$ return k.
   - 3.3 Else

     w_{min} = minimum cost of any clause in κ.

     $mincost = mincost + w_{min}$

     For each soft clause $C_i \in \kappa$ we update Φ
     - 3.3.1 Replace C_i by the two new copies C_i^1 and C_i^2
     - 3.3.2 C_i^1 has a reduced cost $cost(C_i) - w_{min}$
     - 3.3.3 C_i^2 has a **new** b-variable b_i^K and $cost(C_i^2) = w_{min}$

     Add hard clauses $CNF \sum b_i^K = 1$ to Φ.

§3.4. WPM1

- We know that we have to falsify some soft clause in κ. The minimum cost we have to incur is w_{min}.
- The new linear constraint says that at least one of the C_i^2 clauses must be blocked. All have cost w_{min}. So *mincost* must be increased by w_{min}.
- If we incur w_{min} we still have left over weight in C_i when $cost(C_i) > w_{min}$. This is captured in the C_i^1 clauses. (These clauses can generate further cores)
- Can add multiple b-variables to a clause like Fu & Malik

§3.4. WPM2

- Initially add a blocking variable b_i to each soft clause C_i.
- These clauses are never duplicated or changed (no further b-variables will be added).
- Maintains disjoint sets of cores—the cores are grouped into sets where the cores in each set share no soft clauses with the cores in another set.

§3.4. WPM2

- Instead, it maintains a much more complex set of pseudo-boolean constraints over each set of cores. These constraints are over the *b*-variables of the clauses in these cores.
  1. Maintains a lower bound on the *b*-variables in each discovered core—to ensure that all already discovered cores are blocked in subsequent iterations.
  2. Also maintains an upper bound on these *b*-variables—to insure that a minimum weight of soft clauses is blocked in each SAT test.
- The new core might span more than one disjoint set—then these sets have to be unioned.
- The new core causes an update to the pseudo-boolean constraints.

§3.4. WPM2

Example

$cost(C_1) = 10$, $cost(C_2) = 4$, $cost(C_3) = 8$, $cost(C_4) = 2$.

1. $\kappa_1 = \{C_1, C_2\}$: Add $10b_1 + 4b_2 \geq 4$ and $10b_1 + 4b_2 \leq 4$.
2. $\kappa_2 = \{C_3, C_4\}$: Add $8b_3 + 2b_4 \geq 2$ and $8b_3 + 2b_4 \leq 2$.
3. $\kappa_3 = \{C_1, C_2, C_3, C_4\}$.
   - 3.1 Previously $\{\{\kappa_1\}, \{\kappa_2\}\}$ were the disjoint sets of cores. Now the disjoint sets become $\{\{\kappa_1, \kappa_2, \kappa_3\}\}$.
   - 3.2 This core is generated even though we have allowed cost 4 from κ_1 and cost 2 from κ_2

$$10b_1 + 4b_2 + 8b_3 + 2b_4 \geq 6 + 1$$

But from $10b_1 + 4b_2 \geq 4$ and $8b_3 + 2b_4 \geq 2$ we can infer

$$10b_1 + 4b_2 + 8b_3 + 2b_4 \geq 12$$

§3.4. WPM2

- Computing the greatest new lower bound is a subset sum problem (NP-Hard but in practice not a bottle neck).
- The two upper bounds $10b_1 + 4b_2 \leq 4$ and $8b_3 + 2b_4 \leq 2$ are then replaced by $10b_1 + 4b_2 + 8b_3 + 2b_4 \leq 12$

§3.4 Sequence of SAT Instances

- In WPM1 the theory can explode due to duplicating clauses.
- In WPM2 the theory remains small, but the linear constraints are much more complex (they have non-unit coefficients). As disjoint sets of cores become unioned, these constraints also become large.
  - SAT is not good at arithmetical reasoning.
- BINCD ("Core-guided binary search algorithms for maximum satisfiability," Heras, Morgado, Marques-Silva (AAAI 11)) extends WPM2 by doing binary search over the cost of each disjoint set of cores rather than strictly incrementing the lower bound.

§3.5 Hybrid SAT/MIPS approach

The last approach for solving MAXSAT we present is the hybrid approach MAXHS.

"Solving MAXSAT by Solving a Sequence of Simpler SAT Instances", Davies and Bacchus (CP-2011).

"Solving MAXSAT by Decoupling Optimization and Satisfaction", Davies PhD thesis (2013)

Observation (1)

*Given any collection of cores, $\mathcal{K}$, and a truth assignment π s.t. $\pi \models hard(\mathcal{F})$, let $hs(\pi)$ be the set of clauses falsified by π. Then $hs(\pi)$ is a **hitting set** of $\mathcal{K}$, i.e.*

$$\forall \kappa \in \mathcal{K} : hs(\pi) \cap \kappa \neq \emptyset$$

§3.5 Hybrid SAT/MIPS approach

Example

$$\mathcal{K} \begin{cases} \kappa_1 = & \{(x,1),(\neg x,1)\} \\ \kappa_2 = & \{(y,1),(\neg y \vee \neg x,1),(x,1)\} \\ \kappa_3 = & \{(a,1),(\neg a \vee b,1),(\neg b \vee \neg a,1)\} \end{cases}$$

- κ_1, κ_2, and κ_3 are cores

§3.5 Hybrid SAT/MIPS approach

Example

$$\mathcal{K} \begin{cases} \kappa_1 = & \{(x,1),(\neg x,1)\} \\ \kappa_2 = & \{(y,1),(\neg y \vee \neg x,1),(x,1)\} \\ \kappa_3 = & \{(a,1),(\neg a \vee b,1),(\neg b \vee \neg a,1)\} \end{cases}$$

- κ_1, κ_2, and κ_3 are cores
- $\pi = \{x, y, a, b, c\}$ falsifies $(\neg x, 1) \in \kappa_1$, $(\neg y \vee \neg x, 1) \in \kappa_2$ and $(\neg b \vee \neg a, 1) \in \kappa_3$
- the set of clauses falsified by π is a hitting set for $\mathcal{K}$

§3.5 Hybrid SAT/MIPS approach

Definition

A **Minimum Cost Hitting Set (MCHS)** of a collection of cores $\mathcal{K}$ is a hitting set for $\mathcal{K}$ such that all other hitting sets have greater or equal cost.

- The cost of a set of clauses S is the sum of the costs of the clauses it contains: $cost(S) = \sum_{C \in S} cost(C)$.
- The cost of a truth assignment π is the cost of the set of clauses it falsifies: $cost(\pi)$.

§3.5 Hybrid SAT/MIPS approach

Example

$$\mathcal{K} \begin{cases} \kappa_1 = & \{(x,1),(\neg x,1)\} \\ \kappa_2 = & \{(y,1),(\neg y \vee \neg x,1),(x,1)\} \\ \kappa_3 = & \{(a,1),(\neg a \vee b,1),(\neg b \vee \neg a,1)\} \end{cases}$$

- $hs_{min} = \{(x,1),(\neg a \vee b,1)\}$ is a MCHS of $\mathcal{K}$
- $cost(MCHS(\mathcal{K})) = 2$

§3.5 Hybrid SAT/MIPS approach

Theorem

If $\mathcal{K}$ is any collection of cores of the MAXSAT *problem $\mathcal{F}$, and $\pi \models \mathcal{F} \setminus MCHS(\mathcal{K})$*

then π is an optimal truth assignment for $\mathcal{F}$.

§3.5 Hybrid SAT/MIPS approach

1. Input MAXSAT CNF $\mathcal{F}$
2. $\mathcal{K} = \{\}$
3. While *true*
   - 3.1 $hs = MCHS(\mathcal{K})$
   - 3.2 $(\kappa, SAT?) = \text{SAT}(\mathcal{F} \setminus hs)$
   - 3.3 If $SAT?$ return $cost(hs)$.
   - 3.4 Else $\mathcal{K} = \mathcal{K} \cup \{\kappa\}$.

§3.5 Hybrid SAT/MIPS approach

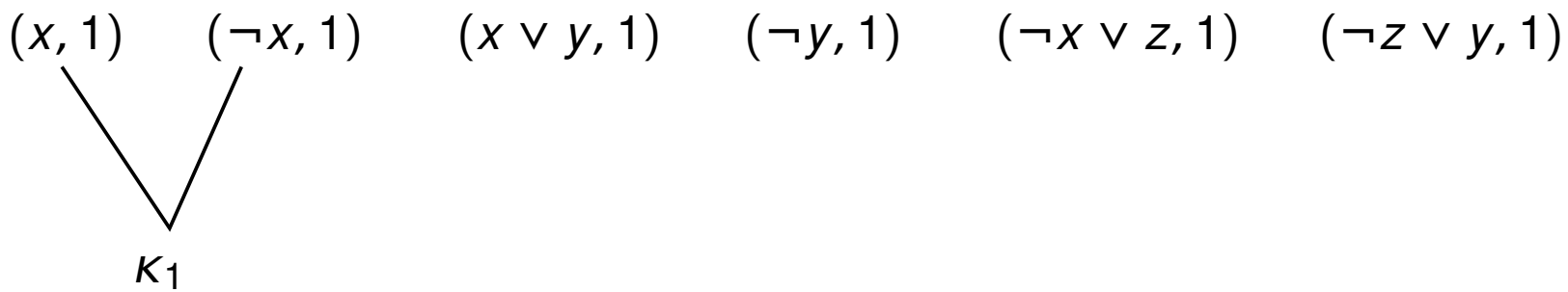
- We use CPLEX to compute $MCHS(\mathcal{K})$.
- No pseudo-boolean constraints are added to the SAT instance, all numeric reasoning with the weights is done in CPLEX
- The SAT instances are easier than the input $\mathcal{F}$—always remove clauses from $\mathcal{F}$.
- The simple theorem proves correctness
- The approach is closely related to Karp's idea of implicit hitting set problems, and can also be viewed as being a logic based Benders decomposition for MAXSAT (Hooker).

§3.5 Hybrid SAT/MIPS approach

$(x, 1)$ $(\neg x, 1)$ $(x \vee y, 1)$ $(\neg y, 1)$ $(\neg x \vee z, 1)$ $(\neg z \vee y, 1)$

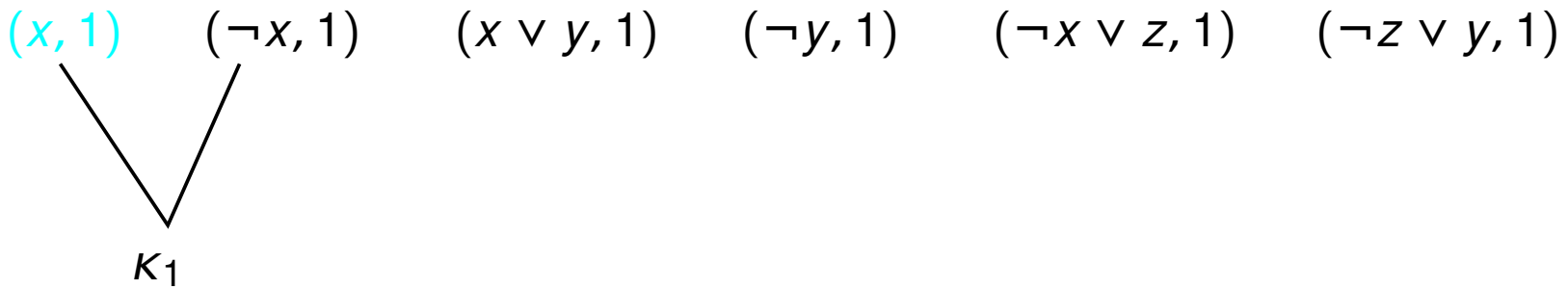
- What is $\mathit{mincost}(\mathcal{F})$?

§3.5 Hybrid SAT/MIPS approach



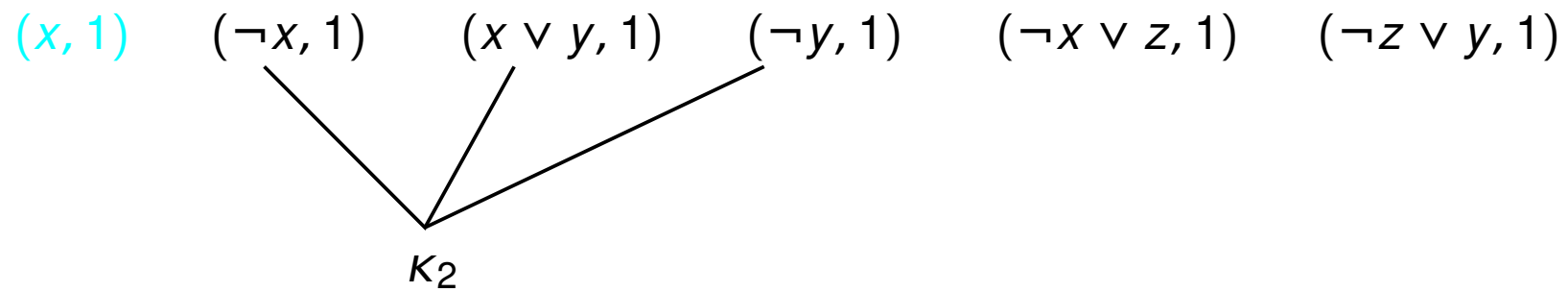
- SAT-Solver($\mathcal{F}$) returns (UNSAT, κ_1)

§3.5 Hybrid SAT/MIPS approach



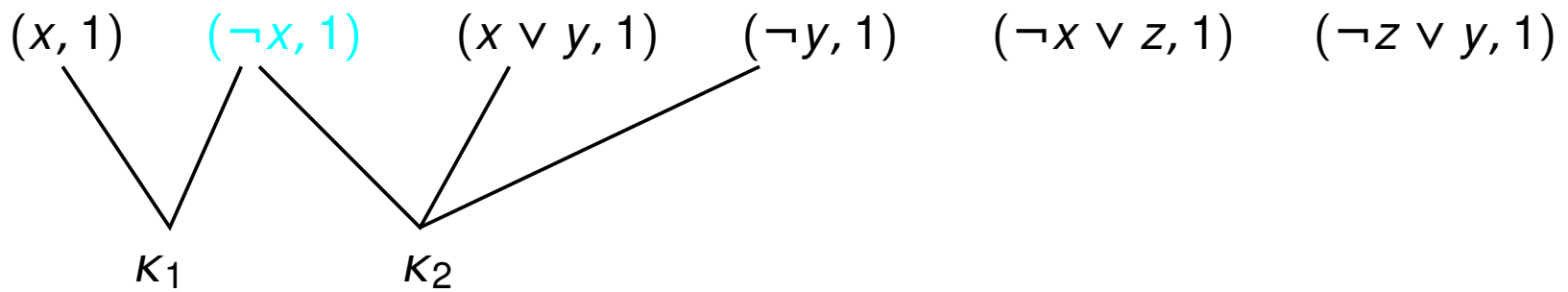
- $\mathrm{MCHS}(\{\kappa_1\}) = 1$

§3.5 Hybrid SAT/MIPS approach



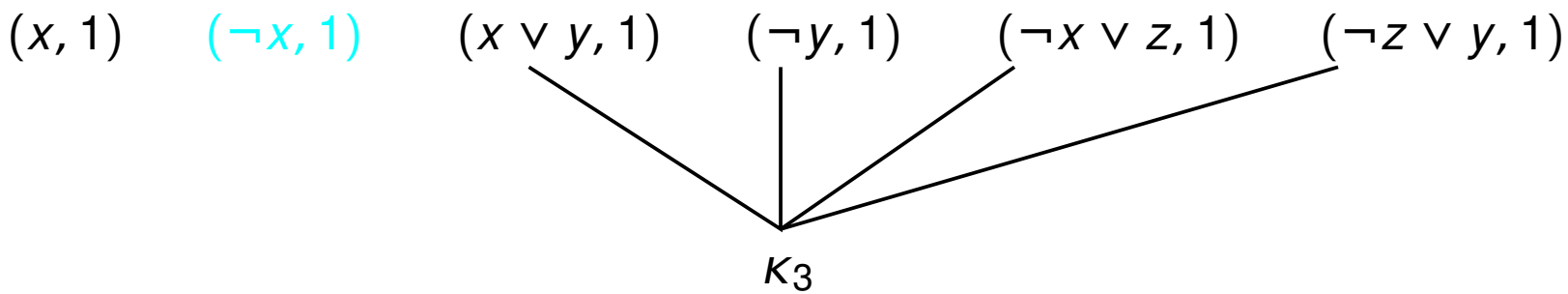
- SAT-Solver($\mathcal{F} \setminus \{(x, 1)\}$) returns (UNSAT, κ_2)

§3.5 Hybrid SAT/MIPS approach



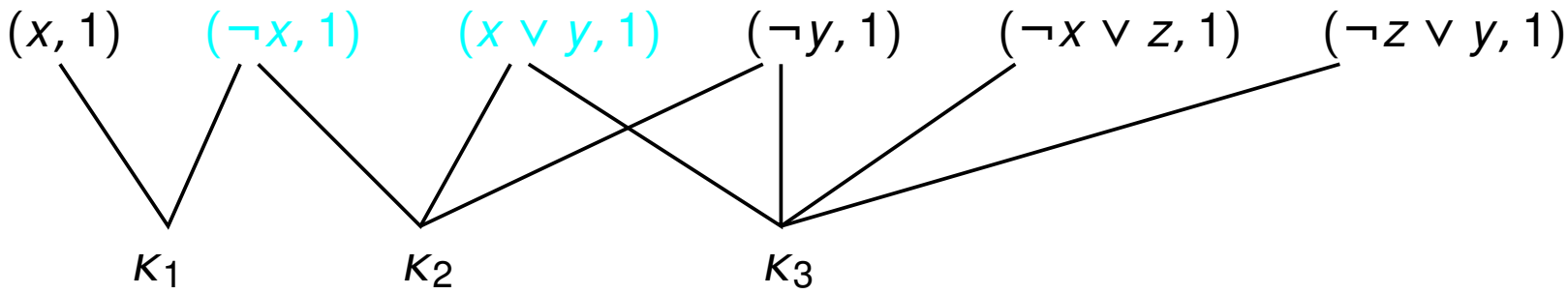
- $\mathrm{MCHS}(\{\kappa_1, \kappa_2\}) = 1$

§3.5 Hybrid SAT/MIPS approach



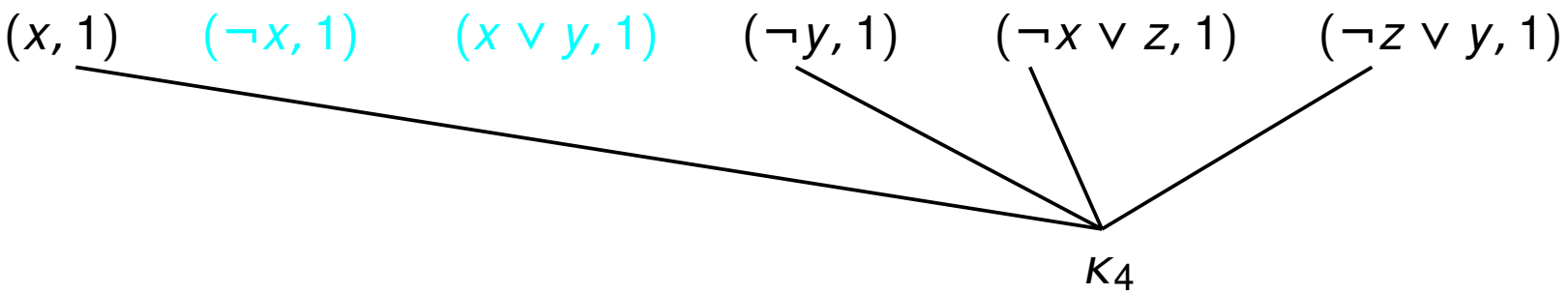
- SAT-Solver($\mathcal{F} \setminus \{(\neg x, 1)\}$) returns (UNSAT, κ_3)

§3.5 Hybrid SAT/MIPS approach



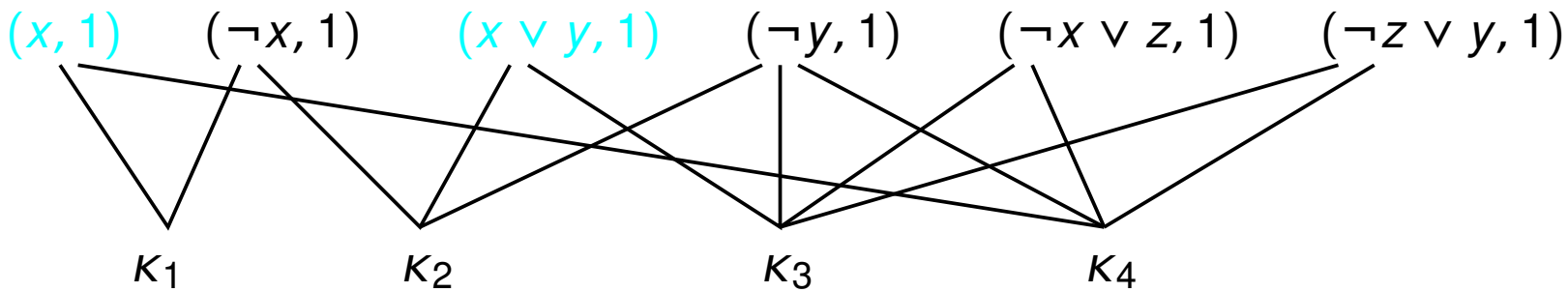
- $\mathrm{MCHS}(\{\kappa_1, \kappa_2, \kappa_3\}) = 2$

§3.5 Hybrid SAT/MIPS approach



- SAT-Solver($\mathcal{F} \setminus \{(\neg x, 1), (x \vee y, 1)\}$) returns (UNSAT, κ_4)

§3.5 Hybrid SAT/MIPS approach



- $\mathrm{MCHS}(\{\kappa_1, \kappa_2, \kappa_3, \kappa_4\}) = 2$

§3.5 Hybrid SAT/MIPS approach

$(x, 1)$ $(\neg x, 1)$ $(x \vee y, 1)$ $(\neg y, 1)$ $(\neg x \vee z, 1)$ $(\neg z \vee y, 1)$

- SAT-Solver($\mathcal{F} \setminus \{(x, 1), (x \vee y, 1)\}$) returns (SAT, π) where $\pi = \{\neg x, \neg y, \neg z\}$ is a solution
- Theorem $\Rightarrow$ π is an optimal truth assignment for $\mathcal{F}$

§3.5 Behaviour of MaxHS

- MaxHS produces a lower bound on $\mathit{mincost}(\mathcal{F})$ at every iteration (i.e., $\mathit{cost}(\mathit{MCHS}(\mathcal{K}))$)
- The lower bound may plateau (i.e., not increase) even though $\mathcal{K}$ is being augmented by more cores
- The SAT solver must refute the formula $\mathcal{F} \setminus \mathit{hs}$ at each iteration
- The MIP solver must find the MCHS of $\mathcal{K}$ at each iteration
- Therefore, there are three potential sources of exponential runtime:
  1. The SAT solving time
  2. The MIP solving time
  3. The number of iterations (i.e., the number of cores required). In the worst case $\mathcal{K}$ must contain an exponential number of cores.

§3.5 Improving MaxHS

- The hitting set problem has additional structure.
  - If $C_1 = (x, y, 3)$ and $C_2 = (\neg x, z, 10)$, C_1 and C_2 cannot both be falsified by any truth assignment.
  - The hard clauses might imply that any truth assignment falsifying C_1 must also falsify C_2, etc.
  - Hittings sets of the cores that violate logical conditions like cannot be optimal solutions.

  To address this we can use the SAT solver to derive additional constraints on the hitting set problem and give these constraints to CPLEX.
- Now CPLEX is no longer solving a pure hitting set problem.

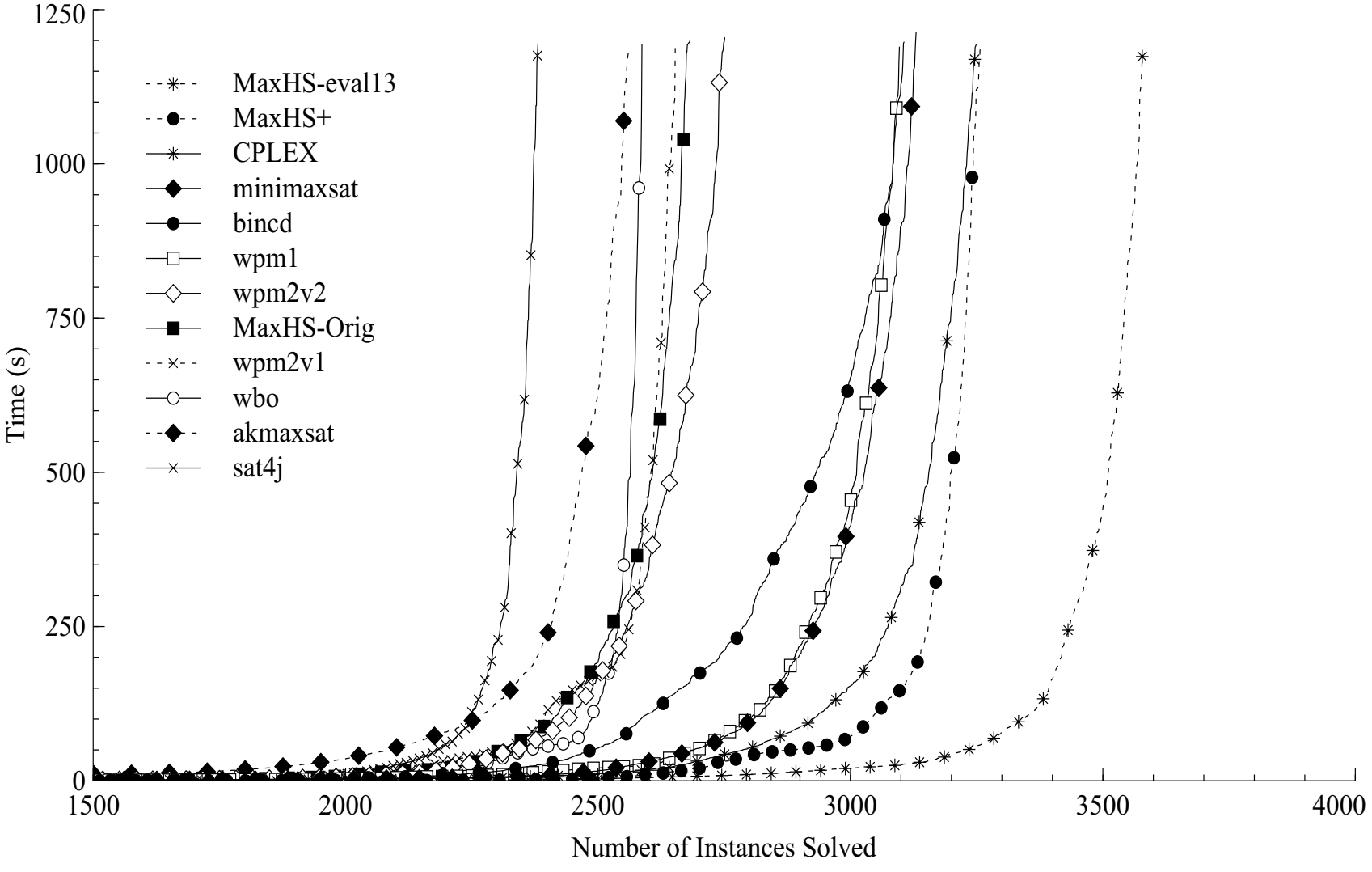
§3.5 Improving MaxHS

- CPLEX is powerful but expensive. It pays to call it fewer times but with harder problems each time.
  - After adding a new core to $\mathcal{K}$, instead of computing a optimal hitting set of $\mathcal{K}$ we compute a greedy or incremental hitting set hs^{-}.
  - We test $\mathcal{F} \setminus hs^{-}$ to see if it is SAT.
    1. If SAT we now have to compute an optimal hitting set of $\mathcal{K}$. For termination to be correct we need an optimal hitting set.
    2. If UNSAT we have a new core we can add to $\mathcal{K}$ without having to call CPLEX.
  - As a result many cores are added to $\mathcal{K}$ before each new call to CPLEX.

§4. Empirical Comparison (as of 2013)

- There is an annual MAXSAT evaluation but the results are not so useful due to machine limitations (the experiments were run on machines with only 500MB RAM).
- Davies in her Ph.D work ran experiments on all 4502 Crafted and Industrial instances from the 2006-2012 MAXSAT Evaluations, an include problems from a diversity of applications
- Solvers tested
  - Branch and Bound solvers: akmaxsat, minimaxsat
  - Core-based solvers: wpm1, wpm2, sat4j, bincd, wbo
  - MIP solver: CPLEX 12.2
  - MAXHS (various versions)
- The data presented here show the number of problems solved under resource limits of 1200 CPU seconds and 2.5GB or RAM.

§4. Empirical Performance



§4. Empirical Performance

Category (#)	mini	bincd	cplex	MaxHS 2011	MaxHS seed	MaxHS nonopt	MaxHS nonopt seed
Crafted (1960)	**1493**	855	1470	847	1092	1137	1374
Industrial (2542)	1637	**2251**	1779	2149	2165	2160	2224
Total (4502)	3130	3106	3249	2996	3257	3297	**3598**

§5. Conclusions

- There are a wide variety of applications of MAXSAT in planning.
  - Planning is based on a logical representation.
  - Typically the modifications to SAT based encodings to obtain MAXSAT encodings are very natural.
- MAXSAT solvers continue to improve and are most likely to be more effective than ad-hoc solutions.