# Combining Linear Programming and Satisfiability Solving for Resource Planning[*†]

**Steven A. Wolfman** **Daniel S. Weld**
Department of Computer Science & Engineering
University of Washington, Box 352350
Seattle, WA 98195–2350 USA
{wolf, weld}@cs.washington.edu

## Abstract

Compilation to boolean satisfiability has become a powerful paradigm for solving AI problems. However, domains that require metric reasoning cannot be compiled efficiently to SAT even if they would otherwise benefit from compilation. We address this problem by introducing the LCNF representation that combines propositional logic with metric constraints. We present LPSAT, an engine that solves LCNF problems by interleaving calls to an incremental Simplex algorithm with systematic satisfaction methods and benefits from both Artificial Intelligence and Operations Research techniques. We describe a compiler that converts metric resource planning problems into LCNF for processing by LPSAT. The experimental section of the paper explores several optimizations to LPSAT, including learning from constraint failure and randomized cutoffs.

## 1 Introduction

Recent advances in boolean satisfiability (SAT) solving technology have rendered large, previously intractable problems quickly solvable [Crawford and Auton, 1993; Selman *et al.*, 1996; Cook and Mitchell, 1997; Bayardo and Schrag, 1997; Li and Anbulagan, 1997; Gomes *et al.*, 1998]. SAT solving has become so successful that many other difficult tasks are being compiled into propositional form to be solved as SAT problems. For example, SAT-encoded solutions to graph coloring, planning, and circuit verification are among the fastest approaches to these problems [Kautz and Selman, 1996;

[*]This paper is based on a previous paper appearing in the proceedings of IJCAI-99. We thank people who provided code, help, and discussion: Greg Badros, Alan Borning, Corin Anderson, Mike Ernst, Zack Ives, Subbarao Kambhampati, Henry Kautz, Jana Koehler, Tessa Lau, Denise Pinnel, Rachel Pottinger, Bart Selman, Dave Smith, and blind reviewers. This research was funded in part by Office of Naval Research Grant N00014-98-1-0147, by the ARCS foundation Barbara and Tom Cable fellowship, by National Science Foundation Grants IRI-9303461 and IIS-9872128, and by a National Science Foundation Graduate Fellowship.

[†]Submitted for review to KER.

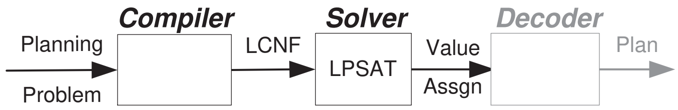

Figure 1: Data flow in the demonstration resource planning system; space precludes discussion of the grey components.

Selman *et al.*, 1997]. These SAT encodings succeed because they compile to a simple yet expressive target language and take advantage of rapidly progressing solution techniques.

However, many real-world tasks have a metric aspect. For instance, resource planning, temporal planning, scheduling, and analog circuit verification problems all require reasoning about real-valued quantities. Unfortunately, metric constraints are difficult and in some cases impossible to express in SAT encodings. Hence, a target language and solution system that could efficiently handle both metric constraints and propositional formulas would yield a powerful substrate for handling AI problems.

This paper introduces a new language, LCNF, which combines the expressive power of propositional logic with that of linear equalities and inequalities. This combination is still less expressive than an Integer Linear Programming (ILP) encoding, but it encourages a careful division of logical and metric reasoning and combines the strengths of constraint satisfaction and Linear Programming (LP). We argue that LCNF provides an ideal target language into which a compiler might translate tasks that combine logical and metric reasoning.

Moreover, we present an architecture for solving LCNF problems that takes advantage of the powerful existing technologies in both of these areas. We describe the LPSAT LCNF solver, which implements this architecture. We also present a number of enhancements and alternatives to the architecture. These include incremental updates to the constraint set, speedup learning via conflict set (nogood) construction in the LP engine, and an optimizing (as opposed to satisficing) variant of LPSAT.

Finally, to demonstrate the utility of the LCNF approach in a concrete domain, we present a fully implemented compiler for resource planning problems. Figure 1 shows how the components fit together: a compiler translates the planning problem into LCNF, the LPSAT system solves the LCNF problem, and a decoder translates the truth/real-value assignment into a plan. The performance of the system is impressive: LPSAT solves large resource planning problems (encoded in a variant of the PDDL language [McDermott, 1998] based on the metric constructs used by metric IPP [Koehler, 1998]), including a metric version of the ATT Logistics domain [Kautz and Selman, 1996].

## 2 The LCNF Formalism

The LCNF representation combines a propositional logic formula with a set of metric constraints in a style similar to that proposed by Hooker et al. [Hooker *et al.*, 1999]. Truth assignments to the boolean satisfiability portion of the problem define the metric constraint set, and both the satisfiability and metric portions

| | | |
|---|---|---|
| $\mathit{MaxLoad} \Rightarrow$ `(load ≤ 30)` | | ; Statements |
| $\mathit{MaxFuel} \Rightarrow$ `(fuel ≤ 15)` | | ; defining |
| $\mathit{MinFuel} \Rightarrow$ `(fuel ≥ 7 + load / 2)` | | ; triggered |
| $\mathit{AllLoaded} \Rightarrow$ `(load = 45)` | | ; constraints |
| $\mathit{MaxLoad}$ | ; Triggers for load and | |
| $\mathit{MaxFuel}$ | ; fuel limits are unit | |
| $\mathit{Deliver}$ | ; The goal is unit | |
| $\neg \mathit{Move} \vee \mathit{MinFuel}$ | ; Moving requires fuel | |
| $\neg \mathit{Move} \vee \mathit{Deliver}$ | ; Moving implies delivery | |
| $\neg \mathit{GoodTrip} \vee \mathit{Deliver}$ | ; A good trip requires | |
| $\neg \mathit{GoodTrip} \vee \mathit{AllLoaded}$ | ; a full delivery | |

Figure 2: Portion of a tiny LCNF logistics problem (greatly simplified from compiler output). A truck with load and fuel limits makes a delivery but is too small to carry all load available (the *AllLoaded* constraint). *Italicized* variables are boolean-valued; `typeface` are real.

must be solved to solve the entire LCNF problem[1].

The key to the encoding is the simple but expressive concept of triggers — each propositional variable may *trigger* a constraint; this constraint is then enforced whenever the variable's truth assignment is `true`. A variable with an associated constraint is called a *trigger variable.* Using this framework, we can construct logical clauses that simulate constraints triggered by `false` assignments, multiple constraints triggered by a single variable, and other more complex triggering schemes.

Formally, an LCNF *problem* is a five-tuple $\langle \mathcal{R}, \mathcal{V}, \Delta, \Sigma, \mathcal{T} \rangle$ in which $\mathcal{R}$ is a set of real-valued variables, $\mathcal{V}$ is a set of propositional variables, $\Delta$ is a set of linear equality and inequality constraints over variables in $\mathcal{R}$, $\Sigma$ is a propositional formula in CNF over variables in $\mathcal{V}$, and $\mathcal{T}$ is a function from $\mathcal{V}$ to $\Delta$ which establishes the constraint triggered by each propositional variable. We require that $\Delta$ contain a special `null` constraint that is vacuously true, and this is used as the $\mathcal{T}$-value for a variable in $\mathcal{V}$ to denote that it triggers no constraint. Moreover, for each variable $v$ we define $\mathcal{T}(\neg v) = \texttt{null}$.

Under this definition, an assignment to an LCNF problem is a mapping, $\varphi$, from the variables in $\mathcal{R}$ to real values and from the variables in $\mathcal{V}$ to truth values. Given an LCNF problem and an assignment, the *set of active constraints* is $\{c \in \Delta | \exists v \in \mathcal{V} \;\; \varphi(v) = \texttt{true} \wedge \mathcal{T}(v) = c\}$. We say that an assignment *satisfies* the LCNF problem if and only if it makes at least one literal true in each clause of $\Sigma$ and satisfies the set of active constraints. A partial assignment to an LCNF problem is a mapping from variables in $\mathcal{V}$ to values in the domain $\{\texttt{true}, \texttt{false}, \texttt{unassigned}\}$. We say that a partial assignment $\vartheta$ is *inconsistent* with respect to an LCNF problem if there is no assignment $\varphi$ that satisfies the problem such that for all $v \in \mathcal{V}$ either $\varphi(v) = \vartheta(v)$ or $\vartheta(v) = \texttt{unassigned}$. Intuitively then, an assignment is satisfying if it makes each propositional clause true and triggers a consistent constraint set; a partial assignment is inconsistent if it cannot be extended to form a satisfying assignment.

[1]Our current LCNF specification does not define any function for measuring the quality of the solution; so, an LCNF solver need not be optimizing.

Figure 2 shows a fragment of a very simple LCNF problem: a truck, which carries a maximum load of 30 and fuel level of 15, can make a *Delivery* by executing the *Move* action. We discuss later why it cannot have a *GoodTrip*.

## 3 The LPSAT Solver

The LPSAT architecture uses a systematic SAT solver as the controlling component of the engine and makes calls to an LP system. The LPSAT algorithm is very similar to the DPLL algorithm for solving boolean satisfiability problems [Davis *et al.*, 1962]. The key difference is in the definitions of "satisfying assignment" and "inconsistent partial assignment". As described in the previous section, each of these concepts in an LCNF problem refers to the active constraint set; however, in a boolean satisfiability problem, the solver considers only the CNF formula when checking for satisfiability. We now describe how we alter LPSAT's SAT component to conform to LCNF's definitions of satisfying and inconsistent.

An assignment to an LCNF statement is satisfying only if its activated constraint set is consistent. So, to handle LCNF problems, the SAT solver's check for satisfaction of its LCNF statement must be modified. The solver still checks if the boolean portion of the problem is satisfied; however, if the boolean portion is satisfied, the solver constructs the active constraint set (according to the truth values of the trigger variables) and checks for consistency with the LP solver. Only if the constraint set is consistent does the solver report satisfaction; otherwise, the assignment is inconsistent. In addition, when actually reporting a solution (as opposed to simply reporting satisfiability), the SAT solver must return values for its propositional variables and query the LP engine for consistent values for the metric variables.

A partial assignment to an LCNF statement is inconsistent if it activates an inconsistent constraint set. To accomodate this definition, the SAT system's check for inconsistency must — if the boolean portion of the problem is found consistent — construct the activated constraint set, check for inconsistency with the LP solver, and return the result. If a trigger variable is unassigned, its constraint should *not* be added to the active constraint set (since an extension of the partial assignment may or may not activate that constraint).

We implemented the LPSAT engine by modifying the RelSAT satisfiability solver [Bayardo and Schrag, 1997] as described above and combining it with the Cassowary constraint solver [Borning *et al.*, 1997; Badros and Borning, 1998]. We chose RelSAT because it was the best available systematic satisfiability solver, and its code was well-structured. RelSAT — like all DPLL-style solvers — performs a systematic, depth-first search through the space of partial truth assignments. It also incorporates powerful learning optimizations which make it competitive with the best of the non-systematic SAT solvers. We chose Cassowary for its efficiency and because it implements incremental Simplex and so supports and quickly responds to small changes in its constraint set.

In addition to modifying RelSAT's check for satisfying and inconsistent assignments, we also had to weaken the pure literal elimination rule. In a CNF problem, pure literal elimination may eliminate certain solutions, but it preserves satisfiability. However, in an LCNF problem, pure literal elimination may not preserve satisfiability because a pure positive trigger variable may trigger an inconsistent constraint; so, setting that variable to true would make the

Procedure LPSAT(LCNF problem: $\langle \mathcal{R}, \mathcal{V}, \Delta, \Sigma, \mathcal{T} \rangle$)
If $\exists$ an empty clause in $\Sigma$ or INC?($\Delta$), return $\{\perp\}$.
Else if $\Sigma$ is empty, return SOLVE($\Delta$).
Else if $\exists$ a pure literal $u$ in $\Sigma$ and $\mathcal{T}(u) = \texttt{null}$,
    return $\{u\} \cup \mathrm{LPSAT}(\langle \mathcal{R}, \mathcal{V}, \Delta, \Sigma(u), \mathcal{T} \rangle)$.
Else if $\exists$ a unit clause $\{u\}$ in $\Sigma$,
    return $\{u\} \cup \mathrm{LPSAT}(\langle \mathcal{R}, \mathcal{V}, \Delta \cup \mathcal{T}(u), \Sigma(u), \mathcal{T} \rangle)$.
Else
    Choose a variable, $v$, mentioned in $\Sigma$.
    Let $\mathcal{A} = \mathrm{LPSAT}(\langle \mathcal{R}, \mathcal{V}, \Delta \cup \mathcal{T}(v), \Sigma(v), \mathcal{T} \rangle)$.
    If $\perp \notin \mathcal{A}$, return $\{v\} \cup \mathcal{A}$.
    Else, return $\{\neg v\} \cup \mathrm{LPSAT}(\langle \mathcal{R}, \mathcal{V}, \Delta, \Sigma(\neg v), \mathcal{T} \rangle)$.

Figure 3: Core LPSAT algorithm (without learning). INC? denotes a check for constraint inconsistency; SOLVE returns constraint variable values. $\mathcal{T}(u)$ returns the constraint triggered by $u$ (possibly `null`). $\Sigma(u)$ denotes the result of setting literal $u$ true in $\Sigma$ and simplifying. The DPLL algorithm is similar but makes no reference to $\mathcal{R}$, $\Delta$, trigger variables, inconsistency checks, or metric constraint solves.

problem unsatisfiable. In order to maintain the satisfiability-preserving property of pure literal elimination, we never consider trigger variables to be pure positive[2].

In general, any heuristic which, like the pure literal elimination rule, preserves satisfiability but not individual solutions may need to be modified to work with LCNF problems. However, heuristics which, like unit propagation, prune only inconsistent assignments from the search space will be applicable to LCNF problems without change[3]. Heuristics which do not prune the search space but merely guide search do not *need* to be modified, but the heuristic may benefit from considering the constraints. For example, we altered RelSAT's variable scoring heuristic for choosing splitting variables at branch points to consider whether a potential split variable is a trigger.

Figure 3 displays pseudocode for our LPSAT algorithm. The algorithm is based on DPLL [Davis *et al.*, 1962]; it performs a depth first search through the space of partial truth assignments. The search backtracks if it reaches an inconsistent partial assignment and succeeds if it finds a satisfying assignment. The pure literal and unit propagation heuristics guide search. Although our discussion and this pseudocode are targeted at systematic solvers, we feel that other kinds of SAT solvers (*e.g.*, stochastic solvers) could be similarly adapted for use in an LCNF solver.

These simple changes will result in a functional LCNF engine; however, more elaborate communication between the SAT and LP modules can exploit the strengths of each module and dramatically improve the performance and ca-

[2]This restriction falls in line with the pure literal elimination rule if we form implicit clauses describing the triggering action. For instance, the implicit clause for the trigger *MaxLoad* $\Rightarrow$ `(load` $\leq$ `30)` from Figure 2 would be $\neg$*MaxLoad* $\vee$ `(load` $\leq$ `30)` and would introduce a negated instance of the trigger variable, *MaxLoad*.

[3]This is because the set of satisfying assignments for an LCNF problem is a subset of the set of satisfying assignments to its CNF portion

pabilities of the whole system. In the next sections, we describe incrementally updating the constraint set, constructing propositional conflict sets (nogoods) from the constraint set, and altering the satisficing SAT system to form an optimizing LPSAT variant.

## 4 Incremental Updates to the Constraint Set

Incremental Simplex systems, including Cassowary, are capable of maintaining and incrementally modifying a constraint set, often much faster than the entire set could be constructed and solved from scratch. In order to take advantage of this behavior in an LCNF engine, the SAT module's procedures for setting (and possibly unsetting) variable values must be modified to notify the LP system of changes in the active constraint set. In particular, when a trigger variable's value becomes `true`, its associated constraint should be added to the LP constraint set; when its value ceases to be `true`, the associated constraint should be removed from the LP constraint set.

However, for current SAT and LP systems, it is generally much faster to set or unset a single propositional variable's value than it is to add or remove a single constraint (even for incremental LP systems). Therefore, it can be valuable to delay commitment of constraints to the constraint set.

With delayed commitment, rather than adding constraints directly to the LP system, the SAT system adds and removes constraints from a cache that maintains the difference between the current constraint set and the set active in the LP system. The LP system never contains constraints that are not part of the active constraint set, but some constraints that are part of the active constraint set will be kept in the cache and not put in the LP system. LPSAT maintains this invariant by actually removing from the LP system any constraint which is deactivated but *was* in the LP system (and not just the cache). Since the active constraint set is always at least as constrained as the constraint set in the LP system, satisfiability checks are complete but not sound and, conversely, inconsistency checks are incomplete but sound. When the SAT system requires a complete inconsistency check or a sound satisfiability check, the cache is entirely added to the LP system. Making a sound (empty cache) satisfiability check before committing to a solution allows the solver to reestablish soundness. By designating the other events that add the cache's constraints to the LP system, an LCNF engine can strike a balance between boolean and metric reasoning time.

These techniques — both incremental updates and delayed commitment — are implemented in the LPSAT system. LPSAT dumps its cache each time it reaches a branch point.

## 5 Learning from Metric Constraint Conflicts

LPSAT inherits methods for speedup learning from RelSAT [Bayardo and Schrag, 1997]. LPSAT's depth-first search of the propositional search space creates a partial assignment to the boolean variables. When the search is forced to backtrack, it is because the partial assignment is inconsistent with the LCNF problem. LPSAT identifies an inconsistent subset of the truth assignments in the partial assignment, a *conflict set*, and uses this subset to enhance its reasoning in two ways. First, since making the truth assignments represented in the conflict set leads inevitably to failure, LPSAT can learn a clause disallowing

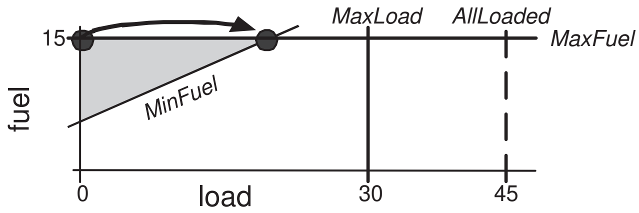


Figure 4: Graphical depiction of the constraints from Figure 2. The shaded area represents solutions to the set of solid-line constraints. The dashed *AllLoaded* constraint causes an inconsistency.

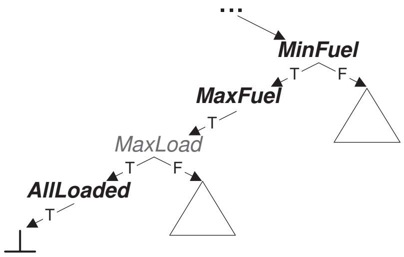


Figure 5: Possible search tree for the constraints from Figure 2. Each node is labeled with the variable set at that node; branchpoints have true (T) and false (F) branches. $\perp$ indicates an inconsistent constraint set. The bold variables are members of the conflict set.

those particular assignments. For example, in the problem from Figure 2 the constraints triggered by setting *MinFuel*, *MaxFuel*, *MaxLoad*, and *AllLoaded* to true are inconsistent, and *MinFuel*, *MaxFuel*, and *AllLoaded* form a conflict set. So, LPSAT can learn the clause ($\neg MinFuel \vee \neg MaxFuel \vee \neg AllLoaded$). Second, because continuing the search is futile until at least one of the variables in the conflict set has its truth assignment changed, LPSAT can backjump in its search to the deepest branch point from which a conflict set variable received its assignment, ignoring any deeper branch points. Figure 5 shows a search tree in which *MinFuel*, *MaxFuel*, *MaxLoad*, and *AllLoaded* have all been set to true. Using the conflict set containing *MinFuel*, *MaxFuel*, and *AllLoaded*, LPSAT can backjump past the branchpoint for *MaxLoad* to the branchpoint for *MinFuel*, the deepest branchpoint at which a member of the conflict set can be changed.

However, while LPSAT inherits methods to use conflict sets from RelSAT, LPSAT must *produce* those conflict sets for both propositional and constraint failures while RelSAT produces them only for propositional failures. Given a propositional failure LPSAT uses RelSAT's conflict set discovery mechanism unchanged, learning a set based on two of the clauses that led to the contradiction [Bayardo and Schrag, 1997]. However, discovering constraint conflict sets requires a new mechanism.

While LPSAT *could* return the entire partial assignment as a conflict set upon discovering an inconsistency in the active constraint set, paring this set down to a smaller set of assignments yields greater pruning action. Since only the trigger variables with `true` values in the partial assignment add to the active constraint set, only those variables need to be included in the conflict set. We call the resulting set a *global conflict set*. To construct this conflict set, the SAT system queries the LP system to get the constraint set; then, it maps the constraints back to the variables that triggered them.

Just as using the triggers of a global conflict set was an improvement over using the entire partial assignment, using any subset of the global conflict set would be an improvement over using the entire set. The logical extension of this idea is to create a conflict set which itself comprises a set of inconsistent constraints but of which every strict subset *is* consistent. We call such a set a *minimal conflict set*. Discovery of both global and minimal conflict sets is implemented in LPSAT; Section 8.1 presents experimental results which show the effectiveness of conflict sets in speedup learning.

Informally, LPSAT finds a minimal conflict set by identifying only those constraints that are, together, in greatest conflict — causing the most error — with the new constraint. In Figure 4, the constraints *MaxLoad*, *MaxFuel*, and *MinFuel* and the implicit constraints that `fuel` and `load` be non-negative are all consistent; however, with the introduction of the dashed constraint marked *AllLoaded* the constraint set becomes inconsistent. We now discuss how LPSAT discovers the conflicting constraints in this figure and which set it discovers.

When LPSAT adds the *AllLoaded* constraint to Cassowary's constraint set, Cassowary initially adds a "slack" version of the constraint that allows error and is thus trivially consistent with the current constraint set. This error is then minimized by the same routine used to minimize the overall objective function [Badros and Borning, 1998]. In Figure 4, we show the minimization as a move from the initial solution at the upper left corner point to the solution at the upper right corner point of the shaded region. The error in the solution is the horizontal distance from the solution point to the new constraint *AllLoaded*. Since no further progress within the shaded region can be made toward *AllLoaded*, the error has been minimized; however, since the error is non-zero, the strict constraint is inconsistent.

At this point, LPSAT constructs a minimal conflict set using *marker variables* (which Cassowary adds to each original constraint). A marker variable is a variable that was added by exactly one of the original constraints and thus identifies the constraint in any derived equations. LPSAT examines the derived equation that gives the error for the new constraint, and notes that each constraint with a marker variable in this equation contributes to keeping the error non-zero. Thus, all the constraints identified by this equation, plus the new constraint itself, compose a conflict set. The LPSAT technical report [Wolfman and Weld, 1999] proves that this technique returns a minimal conflict set.

In Figure 4 the *MinFuel* and *MaxFuel* constraints restrain the solution point from coming closer to the *AllLoaded* line. If the entire active constraint set were any two of those three constraints, the intersection of the two constraints' lines would be a valid solution; however, there is no valid solution with all three constraints.

Note that another conflict set — *AllLoaded* plus *MaxLoad* — exists. In general, there may be many minimal conflict sets, but our conflict discovery technique can discover only one of these per solve. Using careful modifications to the active constraint set and multiple Cassowary solves, we can find many or all of

Procedure BEST-CONFLICT-SET($\Delta$: constraints $\{1, \ldots, |\Delta|\}$)
Let $\mathcal{M} =$ REV-SORT(MIN-CONFLICT-SET($\Delta$)).
Return BCS-HELPER($\Delta$, $\mathcal{M}$, 1).

Procedure BCS-HELPER( $\Delta$: remaining constraints in increasing order,
$\mathcal{M}$: best conflict set so far in decreasing order,
$i$: integer)
If $i = |\mathcal{M}|$, return $\mathcal{M}$.
Else
    Let $\Delta' = \Delta - \{\mathcal{M}_{i+1}, \ldots, \mathcal{M}_i - 1\}$.
    Let $\mathcal{M}' =$ REV-SORT(MIN-CONFLICT-SET($\Delta'$)).
    If $\mathcal{M}' = \emptyset$,
        return BCS-HELPER($\Delta' \cup \{\mathcal{M}_{i+1}\}, \mathcal{M}, i+1$).
    Else
        return BCS-HELPER($\Delta, \mathcal{M}', i$).

Figure 6: Pseudocode for finding the "best" conflict set for the set $\Delta$. The best set is a minimal conflict set, and its worst element is better than the worst element of all other min. conflict sets, ties broken by comparing successively better pairs of elements. Constraints are numbered from best, 1, to worst, $|\Delta|$. We assume that the constraint $|\Delta|$ caused the inconsistency and so is a member of every minimal conflict set. REV-SORT sorts a set into decreasing order; MIN-CONFLICT-SET finds a minimal conflict set or returns $\emptyset$ if the set is consistent; the set $\mathcal{M}$ is made up of the elements $\{\mathcal{M}_1, \ldots, \mathcal{M}_{|\mathcal{M}|}\}$

the minimal conflict sets. In order to differentiate between these, we may need to use a different metric from the one that led us to use minimal conflict sets over global conflict sets. Using the size of the sets is still an option, but since none of these sets is a subset of any of the others, a smaller set is no longer guaranteed to result in better (or even equally good) pruning of the search.

In order to allow the system to refine the choice of minimal conflict set, we can pass a ranking of the trigger variables (and thus the constraints) from the SAT system to LP system. Using a linear (in the number of active constraints) number of calls to the minimal conflict set discovery mechanism described above, the LP system can construct the minimal conflict set with the highest ranked constraints. The algorithm starts from the lowest ranked constraints and removes them one by one until the set becomes consistent, using the minimal conflict sets constructed at each resolve to guide the search. Once the lowest ranked constraint in the minimal conflict set is established, it permanently includes that constraint and moves on to establishing the next lowest ranked constraint. Pseudocode for this algorithm appears in Figure 6. Note that this is an anytime algorithm; it continually improves its solution but always has a solution available. For RelSAT, a DPLL-style solver, we propose ranking the trigger variables in order of their depth in the search tree with the highest ranked variables appearing nearest to the root[4]. This algorithm has not yet been implemented in the LPSAT engine.

[4] Thanks to Rao Kambhampati for this suggestion.

## 6 LPSAT for Optimization Problems

Since LP systems provide a clear notion of optimality — minimize (or maximize) the value of an objective function over the variables in the problem — it is natural to extend this notion to an LCNF system. Given an objective function over the metric variables, we define an optimal solution to an LCNF problem to be that satisfying solution which yields the minimal value for the objective function. However, choosing the optimal values for the LP variables in a given active constraint set will not necessarily minimize the objective function over all possible active constraint sets. There may be another satisfying assignment to the boolean variables in the problem that activates a constraint set with a better value for the objective function. Therefore, in order to construct an optimizing LCNF solver, the satisficing SAT system must be modified.

LPSAT's systematic SAT engine has the capacity to find every possible solution to a SAT problem by continuing its depth first search even after a solution is found. Using this capacity and keeping track of the optimal solution so far, we can construct an optimizing version of LPSAT. Of course, not every solution needs to be visited; optimizing LPSAT can use a branch and bound strategy to eliminate unpromising search branches. This is because the objective value of a partial assignment is always at least as good as the value of its extensions since, given a constraint set with some value for the objective function, more constraints can never improve that value.

Implementing this modification to the LCNF system requires enhancing the communication between the SAT and LP components; in this case, the SAT component must query the LP component for objective values of each partial (and total) assignment. Also, the LP system must have an objective function on which to base its evaluations. In Cassowary, there is a default objective function; however, in general, an optimizing LCNF system would require support for an objective function in the LCNF language.

Neither these optimizing extensions nor the enhancements to LCNF have yet been implemented for the LPSAT system.

## 7 The Resource Planning Application

In order to demonstrate LPSAT's utility, we implemented a compiler for metric planning domains (starting from a base of IPP's [Koehler *et al.*, 1997] and BLACKBOX's [Kautz and Selman, 1998] parsers) which translates resource planning problems into LCNF form. After LPSAT solves the LCNF problem, a small decoding unit maps the resulting boolean and real-valued assignments into a solution plan (Figure 1). We believe that this translate/solve/decode architecture is effective for a wide variety of problems.

### 7.1 Action Language

Our planning problems are specified in an extension of the PDDL language [McDermott, 1998]; we support PDDL typing, equality, quantified goals and effects, disjunctive preconditions, and conditional effects. In addition, we handle metric values with two new built-in types: *float* and *fluent*. A float's value may not change over the course of a plan, whereas a fluent's value may change from time step to time step. Moreover, we support fluent- and float-valued functions, such as `?distance[Seattle,Durham]`.

```
Action loop_a              Action loop_b
 pre: test fluent1 = 0      pre: test fluent2 = 0
 eff: set fluent2 = 1       eff: set fluent1 = 1
```

Figure 7: Two actions which can execute in parallel, but which cannot be serialized.

Floats and fluents are manipulated with three special built-in predicates: `test`, `set`, and `influence`. `Test` statements are used as predicates in action preconditions; `set` and `influence` are used in effects. As its argument, `test` takes a constraint (an equality or inequality between two expressions composed of floats, fluents, and basic arithmetic operations); it evaluates to true if and only if the constraint holds. `Set` and `influence` each take two arguments: the object (a float or fluent) and an expression. If an action causes a `set` to be asserted, the object's value after the action is defined to be the expression's value before the action. An asserted `influence` changes an object's value by the value of the expression, as in the equation $object := object + expression$; multiple simultaneous `influences` are cumulative in their effect [Falkenhainer and Forbus, 1988].

### 7.2 Plan Encoding

The compiler uses a regular action representation with explanatory frame axioms and conflict exclusion [Ernst *et al.*, 1997]. We adopt a standard fluent model in which time takes nonnegative integer values. State-fluents occur at even-numbered times and actions at odd times. The initial state is completely specified at time zero, including all properties presumed false by the closed-world assumption.

Each `test`, `set`, and `influence` statement compiles to a propositional variable that triggers the associated constraint. Just as logical preconditions and effects are implied by their associated actions, the triggers for metric preconditions and effects are implied by their actions.

The compiler must generate frame axioms for constraint variables as well as for propositional variables, but the axiomatizations are very different. Explanatory frames are used for boolean variables, while for real variables, compilation proceeds in two steps. First, we create a constraint which, if activated, will set the value of the variable at the next step equal to its current value plus all the influences that might act on it (untriggered influences are set to zero). Next, we construct a clause which activates this constraint unless some action actually sets the variable's value.

For a parallel encoding, the compiler must consider certain `set` and `influence` statements to be mutually exclusive. For simplicity, we adopt the following convention: two actions are mutually exclusive if and only if at least one `sets` a variable which the other either `influences` or `sets`.

This exclusivity policy results in a plan which is correct if actions at each step are executed strictly in parallel; however, the actions may not be serializable, as demonstrated in Figure 7. In order to make parallel actions arbitrarily serializable, we would have to adopt more restrictive exclusivity conditions and a less expressive format for our `test` statements.

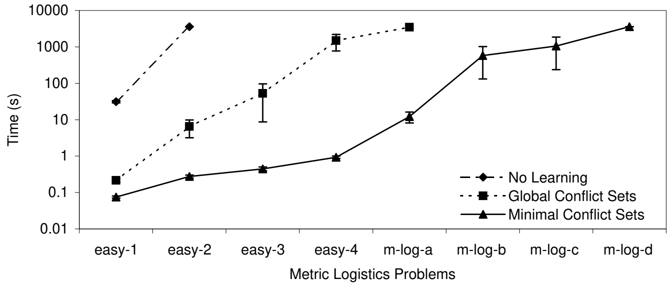


Figure 8: Solution times for three versions of LPSAT in the metric logistics domain. No learning or backjumping is performed in the line marked "No learning." Global conflict sets and minimal conflict sets use progressively better learning algorithms. Note that the final point on each curve reaches the resource cutoff (one hour).

## 8 Experimental Results

There are currently few available metric planners with which to compare LPSAT. The ZENO system [Penberthy and Weld, 1994] is more expressive than our system, but ZENO is unable even to complete *easy-1*, our simplest metric logistics problem. There are only a few results available for Koehler's metric IPP system [Koehler, 1998], and code is not yet available for direct comparisons.

In light of this, this section concentrates on displaying results for LPSAT in an interesting domain and on displaying our heuristics and the benefits of communication between the LP and SAT components. We report LPSAT solve time, running on a Pentium II 450 MHz processor with 128 MB of RAM, averaged over 20 runs per problem, and showing 95 percent confidence intervals. We do not include compile time for the (unoptimized) compiler since the paper's focus is the design and optimization of LPSAT; however, compile time can be substantial (*e.g.*, twenty minutes on *m-log-c*) since it is heavily memory-bound.

We report on a sequence of problems in the metric logistics domain, which includes all the features of the ATT logistics domain [Kautz and Selman, 1996]: airplanes and trucks moving packages among cities and sites within cities. However, our metric version adds fuel and distances between cities; airplanes and trucks both have individual maximum fuel capacities, consume fuel to move (the amount is per trip for trucks and based on distance between cities for airplanes), and can refuel at depots. *m-log-a* through *m-log-d* are the same as the ATT problems *log-a* through *log-d* except for the metric component. *easy-1* through *easy-4* are simplifications of *m-log-a* with more elements retained in the higher numbered problems. We report on experiments with learning as well as two other interesting optimizations.

### 8.1 Learning

The results in Figure 8 demonstrate the improvement in solving times resulting from activating the learning and backjumping facilities which were described in Section 5. Runs were cut off after one hour of solve time (the minimal conflict set technique ran over an hour only twice on *m-log-c* and not at all on easier problems). Without learning or backjumping LPSAT quickly exceeds the

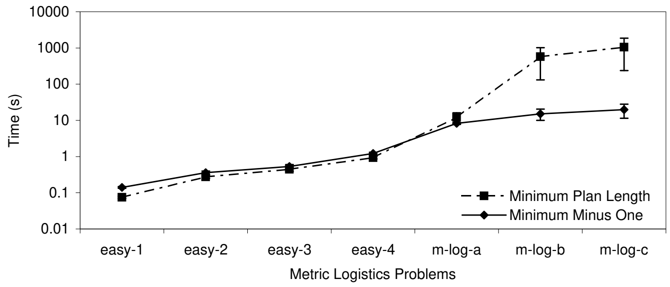


Figure 9: Solution times for LPSAT with minimal conflict sets. The dashed line is the time to find a solution for each problem compiled with the minimum number of steps. The solid line is the time to find that no solution exists for each problem compiled with one fewer steps. LPSAT quickly finds that no solution exists with fewer than the minimum number of steps.

maximum time allotted to it. With learning and backjumping activated using global conflict sets, the solver handles larger problems and runs faster. Our best method, minimal conflict sets, quickly solves even some of the harder problems in the metric logistics domain. Figure 9 shows that the minimal conflict technique runs particularly well when verifying the absence of a solution.

## 8.2 Splitting Heuristic

Line 7 of the LPSAT pseudocode (Figure 3) makes a choice of variable $v$ — called the *splitting variable* — before the recursive call; although we do not need to backtrack over this choice (*i.e.*, the choice is not nondeterministic), a good choice of splitting variable can speed search. We expected the existing RelSAT splitting heuristic to perform poorly because it could not take into account whether a variable is a trigger. This blindness is particularly important since each time the solver modifies a trigger variable, it may call Cassowary, and these calls often dominate runtime. We tried several methods of including information about the trigger variables in the splitting heuristic, including adding and multiplying the score of trigger variables by a user-settable preference value. To our surprise, modifying the score of trigger variables resulted in no significant improvement. These results are inconclusive but may indicate that our compilation of metric planning domains already encodes some information about trigger variables in the structure of the problem which the current heuristic already uses. Further experiments will decide the issue.

## 8.3 Random Restarts

Because LPSAT uses a randomized backtracking algorithm and because early experimental results showed a small percentage of runs far exceeded the median runtime, we experimented with random restarts using a process similar to the one described in [Gomes *et al.*, 1998]. We cut off solving at a deadline — which can be either fixed beforehand or geometrically increasing — and restart the solver with a new random seed.

Figure 10 shows the results of these experiments. We first ran the algorithm

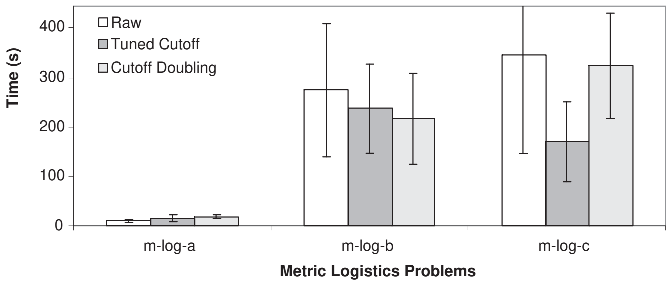


Figure 10: Solution times for two types of random restarts. Tuned cutoff uses raw experimental data to select a constant cutoff. Cutoff doubling starts with a cutoff of one second and doubles it on each run.

twenty times on each problem to produce the "Raw" entries[5]. Then, we calculated the cutoff time that minimized the expected runtime of the system based on these twenty runs. Finally, we reran the problems with this tuned cutoff time to produce the "Tuned Cutoff" data.

While this technique provides some speedup on *m-log-b* and impressive speedup on *m-log-c*, it requires substantial, preliminary research into the difficulty of the problem (in order to determine the appropriate cutoff time). Unless LPSAT is being used repeatedly to solve a single problem or several very similar problems, the process of finding good restart times will dominate overall runtime.

Therefore, we also experimented with a restart system which requires no prior analysis. This "Cutoff doubling" approach sets an initial restart limit of one second and increases that limit by a factor of two on each restart until reaching a solution. We have not yet performed any theoretical analysis of the effectiveness of this technique, but Figure 10 demonstrates a small improvement. More interesting than the average improvement, however, is the fact that this method improved the consistency of the runtimes on the harder problems; indeed, on *m-log-c* five of the twenty "Raw" runs lasted longer than the longest "Cutoff doubling" run.

## 9 Related Work

Limited space precludes a survey of boolean satisfiability algorithms and linear programming methods in this paper. See [Cook and Mitchell, 1997] for a survey of satisfiability and [Karloff, 1991] for a survey of linear programming.

Our work was inspired by the idea of compiling probabilistic planning problems to MAJSAT [Majercik and Littman, 1998]. It seemed that if one could extend the SAT "virtual machine" to support probabilistic reasoning, then it would be useful to consider the orthogonal extension to handle metric constraints. Hooker at al. [Hooker *et al.*, 1999] argue convincingly that Operations Research techniques (such as LP) and Artificial Intelligence techniques (such as SAT solving) could be combined to their mutual benefit, and our system bears this notion out.

[5] All three sets of runs use minimal conflict sets, learning, and backjumping.

Other researchers have combined logical and constraint reasoning, usually in the context of programming languages. CLPR may be thought of as an integration of Prolog and linear programming, and this work introduced the notion of incremental Simplex [Jaffar *et al.*, 1992]. Saraswat's thesis [Saraswat, 1989] formulates a family of programming languages which operate through the incremental construction of a constraint framework.

A variety of recent systems have addressed the issue of integrating metric reasoning into planning. ILPPLAN [Kautz and Walser, 1999] solves planning problems that have been manually encoded as integer linear programs. While integer linear programming (ILP) is more expressive than LCNF, solvers for ILP problems tend to perform poorly on problems which are primarily propositional [Kautz and Walser, 1999]; therefore, LPSAT has the advantage over ILPPLAN on many planning problems, and future solvers for LCNF can continue to exploit advances in SAT engines for solving purely propositional problems. Alternatively, the LPSAT compiler could be used to construct ILPs through a straightforward transformation of the propositional portion to an integer program. Vossen et al. [Vossen *et al.*, 1999] investigate a variety of new encodings and techniques to use ILP to solve planning problems.

CPLAN [van Beek and Chen, 1999] is similar to ILPPLAN but solves planning problems that have been manually encoded as constraint satisfaction problems. While CPLAN's results are promising, no automatic compiler from a planning language to a CPLAN-style CSP exists, and it is unclear how to take advantage of the flexibility of general CSPs.

BLACKBOX uses a translate/solve/decode scheme from planning to satisfiability [Kautz and Selman, 1998]. ZENO is a causal link temporal planner which handles resources by calling an incremental Simplex algorithm within the plan-refinement loop [Penberthy and Weld, 1994]. The GRAPHPLAN [Blum and Furst, 1995] descendant IPP has also been extended to handle metric reasoning in its plan graph [Koehler, 1998].

## 10 Conclusions and Future Work

LPSAT is a promising new technique that combines the strengths of fast boolean satisfiability solving methods with an incremental Simplex algorithm to efficiently handle problems involving both propositional and metric reasoning. This paper describes the following contributions:

- We defined the LCNF formalism for combining boolean satisfiability with linear (in)equalities.
- We implemented the LPSAT solver for LCNF by combining the RelSAT satisfiability solver [Bayardo and Schrag, 1997] with the Cassowary constraint reasoner [Badros and Borning, 1998].
- We developed a variety of optimizations and enhancements for LPSAT: incrementally updating the constraint set, caching constraints, constructing constraint conflict sets, and making an optimizing variant of LPSAT.
- We experimented with three optimizations for LPSAT: adapting the splitting heuristic to trigger variables, adding random restarts, and incorporating learning. Using *minimal conflict sets* to guide learning provided four orders of magnitude speedup.

- We implemented a compiler for resource planning problems. LPSAT's performance with this compiler was much better than that of ZENO [Penberthy and Weld, 1994].

Much remains to be done. We wish to investigate the issue of tuning restarts to problems, including a thorough investigation of exponentially growing resource limits. It would also be interesting to implement an LCNF solver based on a stochastic engine. It would be interesting to investigate precomputing all minimal conflict sets in an LCNF problem and add appropriate learned clauses to the CNF portion of the problem, entirely removing the need for the metric constraints[6]. Finally, we would like to add dynamic backtracking [Ginsberg and McAllester, 1994] to LPSAT in the hopes that it will reduce the number of unnecessary constraint adds and deletes incurred during normal backjumping.

## References


[Badros and Borning, 1998] Greg J. Badros and Alan Borning. The Cassowary Linear Arithmetic Constraint Solving Algorithm: Interface and Implementation. Technical Report 98-06-04, University of Washington, Department of Computer Science and Engineering, June 1998.

[Bayardo and Schrag, 1997] R. Bayardo and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–208, Providence, R.I., July 1997. Menlo Park, Calif.: AAAI Press.

[Blum and Furst, 1995] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1636–1642. San Francisco, Calif.: Morgan Kaufmann, 1995.

[Borning *et al.*, 1997] Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, October 1997.

[Cook and Mitchell, 1997] S. Cook and D. Mitchell. Finding hard instances of the satisfiability problem: A survey. *Proceedings of the DIMACS Workshop on Satisfiability Problems*, pages 11–13, 1997.

[Crawford and Auton, 1993] J. Crawford and L. Auton. Experimental results on the cross-over point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27. Menlo Park, Calif.: AAAI Press, 1993.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *C. ACM*, 5:394–397, 1962.

[Ernst *et al.*, 1997] M. Ernst, T. Millstein, and D. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1169–1176. San Francisco, Calif.: Morgan Kaufmann, 1997.

[Falkenhainer and Forbus, 1988] B. Falkenhainer and K. Forbus. Setting up large scale qualitative models. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 301–306. Menlo Park, Calif.: AAAI Press, August 1988.

[Ginsberg and McAllester, 1994] Matthew L. Ginsberg and David A. McAllester. Gsat and dynamic backtracking. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*. San Francisco, Calif.: Morgan Kaufmann, 1994.


[6]Thanks to Dave Smith for this suggestion.

[Gomes *et al.*, 1998] C.P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 431–437, Madison, WI, July 1998. Menlo Park, Calif.: AAAI Press.

[Hooker *et al.*, 1999] J.N. Hooker, G. Ottosson, E.S. Thorsteinsson, and H. Kim. On integrating constraint propagation and linear programming for combinatorial optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, Florida, July 1999. Menlo Park, Calif.: AAAI Press.

[Jaffar *et al.*, 1992] Joxan Jaffar, Spiro Michaylov, Peter Stuckey, and Roland Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[Karloff, 1991] H. Karloff. *Linear Programming*. Birkhäuser, Boston, 1991.

[Kautz and Selman, 1996] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1194–1201. Menlo Park, Calif.: AAAI Press, 1996.

[Kautz and Selman, 1998] H. Kautz and B. Selman. Blackbox: A new approach to the application of theorem proving to problem solving. In *AIPS98 Workshop on Planning as Combinatorial Search*, pages 58–60. Pittsburgh, Penn.: Carnegie Mellon University, June 1998.

[Kautz and Walser, 1999] H. Kautz and J.P. Walser. State-space planning by integer optimization. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, Florida, July 1999. Menlo Park, Calif.: AAAI Press.

[Koehler *et al.*, 1997] J. Koehler, B. Nebel, J. Hoffmann, and Y Dimopoulos. Extending planning graphs to an ADL subset. In *Proceedings of the Fourth European Conference on Planning*, pages 273–285. Berlin, Germany: Springer-Verlag, Sept 1997.

[Koehler, 1998] J. Koehler. Planning under resource constraints. In *Proceedings of the Thirteenth European Conference on Artificial Intelligence*, pages 489–493. Chichester, UK: John Wiley & Sons, 1998.

[Li and Anbulagan, 1997] C. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 366–371. San Francisco, Calif.: Morgan Kaufmann, August 1997.

[Majercik and Littman, 1998] S. M. Majercik and M. L. Littman. MAXPLAN: a new approach to probabilistic planning. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 86–93. Menlo Park, Calif.: AAAI Press, June 1998.

[McDermott, 1998] Drew McDermott. *PDDL — The Planning Domain Definition Language*. AIPS-98 Competition Committee, draft 1.6 edition, June 1998.

[Penberthy and Weld, 1994] J.S. Penberthy and D. Weld. Temporal planning with continuous change. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. Menlo Park, Calif.: AAAI Press, July 1994.

[Saraswat, 1989] Vijay A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

[Selman *et al.*, 1996] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 26:521–532, 1996.

[Selman *et al.*, 1997] Bart Selman, Henry Kautz, and David McAllester. Computational challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 50–54. San Francisco, Calif.: Morgan Kaufmann, 1997.

[van Beek and Chen, 1999] Peter van Beek and Xinguang Chen. Cplan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, Orlando, Florida, July 1999. Menlo Park, Calif.: AAAI Press.

[Vossen *et al.*, 1999] T. Vossen, M. Ball, A. Lotem, and D. Nau. On the use of integer programming models in ai planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Stockholm, Sweden, Aug 1999. San Francisco, Calif.: Morgan Kaufmann.

[Wolfman and Weld, 1999] S. Wolfman and D. Weld. The LPSAT system and its Application to Resource Planning. Technical Report 99-04-04, University of Washington, Department of Computer Science and Engineering, April 1999.