The LPSAT Engine & its Application to 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 which combines propositional logic with metric constraints. We present LPSAT, an engine which solves LCNF problems by interleaving calls to an incremental simplex algorithm with systematic satisfaction methods. We describe a compiler which 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 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; Selman *et al.*, 1997].

*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, and the 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.

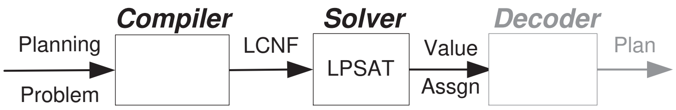

Figure 1: Data flow in the demonstration resource planning system; space precludes discussion of the grey components.

But 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 to express in SAT encodings[1]. Hence, a solver which could efficiently handle both metric constraints and propositional formulae would yield a powerful substrate for handling AI problems.

This paper introduces a new problem formulation, LCNF, which combines the expressive power of propositional logic with that of linear equalities and inequalities. We argue that LCNF provides an ideal target language into which a compiler might translate tasks that combine logical and metric reasoning. We also describe the LPSAT LCNF solver, a systematic satisfiability solver integrated with an incremental Simplex algorithm. As LPSAT explores the propositional search space it updates the set of metric requirements managed by the linear program solver; in turn, Simplex notifies the propositional solver if these requirements become unsatisfiable.

We report on three optimizations to LPSAT: learning and backjumping, adapting LPSAT's core heuristic to trigger variables, and using random restarts. The most effective of these is the combination of learning and backjumping; LPSAT learns new clauses by discovering explanations for failure when a branch of its search terminates. The resulting clauses guide backjumping and constrain future truth assignments. In particular, we show that analysis of the state of the linear program solver is crucial in order to learn effectively from constraint conflicts.

To demonstrate the utility of the LCNF approach, we also present a fully implemented compiler for resource

[1]Encoding each value as a separate boolean variable is a simple but unwieldy solution; bitwise-encodings produce smaller formulae but ones which appear very hard to solve [Ernst *et al.*, 1997].

MaxLoad ⇒ `(load ≤ 30)`	; Statements
MaxFuel ⇒ `(fuel ≤ 15)`	; defining
MinFuel ⇒ `(fuel ≥ 7 + load / 2)`	; triggered
AllLoaded ⇒ `(load = 45)`	; constraints

MaxLoad	; Triggers for load and
MaxFuel	; fuel limits are unit
Deliver	; The goal is unit
¬*Move* ∨ *MinFuel*	; Moving requires fuel
¬*Move* ∨ *Deliver*	; Moving implies delivery
¬*GoodTrip* ∨ *Deliver*	; A good trip requires
¬*GoodTrip* ∨ *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.

planning problems. Figure 1 shows how the components fit together. Their performance 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. 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 *trigger variable*'s truth assignment is `true`.

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, Δ is a set of linear equality and inequality constraints over variables in $\mathcal{R}$, Σ is a propositional formula in CNF over variables in $\mathcal{V}$, and $\mathcal{T}$ is a function from $\mathcal{V}$ to Δ which establishes the constraint triggered by each propositional variable. We require that Δ contain a special `null` constraint which 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, φ, 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 Σ and satisfies the set of active constraints.

Figure 2 shows a fragment of a sample 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

Our first step in constructing the LPSAT engine was to choose solvers to use as the foundation for its metric and propositional solving portions. The choice was motivated by the following criteria:

1. It must be easy to modify the propositional solver in order to support triggers and handle reports of inconsistency from the constraint reasoner.
2. The metric solver must support incremental modifications to the constraint set.
3. Because a Simplex solve is more expensive than setting a single propositional variable's value, the propositional solver should minimize modifications to the constraint set.

These principles led us to implement the LPSAT engine by modifying the RELSAT satisfiability engine [Bayardo and Schrag, 1997] and combining it with the CASSOWARY constraint solver [Borning *et al.*, 1997; Badros and Borning, 1998] using the method described in [Nelson and Oppen, 1979]. RELSAT makes an excellent start for processing LCNF for three reasons. First, it performs a systematic, depth-first search through the space of partial truth assignments; this minimizes changes to the set of active metric constraints. Second, the code is exceptionally well-structured. Third, RELSAT incorporates powerful learning and backjumping optimizations. CASSOWARY is an appropriate Simplex solver for handling LCNF because it was designed to support and quickly respond to small changes in its constraint set.

In order to build LPSAT, we modified RELSAT to include trigger variables and constraints. This required four changes. First, the solver must trigger constraints as the truth assignment changes. Second, the solver must now check for a solvable constraint set to ensure that a truth assignment is satisfying. Third, the solver must report in its solution not only a truth assignment to the boolean variables, but an assignment of real values to the constraint variables[2]. Finally, since even a purely positive trigger variable may (if set to `true`) trigger an inconsistent constraint, pure literal elimination cannot act on *positive* trigger variables[3]. Figure 3 displays pseudocode for the resulting algorithm.

4 Incorporating Learning and Backjumping

LPSAT inherits methods for learning and backjumping from RELSAT [Bayardo and Schrag, 1997]. LPSAT's

[2] While the assignment to the constraint variables is optimal according to CASSOWARY's objective function, it is not guaranteed to be the globally optimal assignment to the real variables by the same measure; a different assignment to the propositional variables might provide a better solution. So, the specific function used is not vital (we use CASSOWARY's default which minimizes the slack in inequalities).

[3] This restriction falls in line with the pure literal elimination rule if we consider the triggers themselves to be clauses. The trigger *MaxLoad* ⇒ `(load ≤ 30)` from Figure 2 would then become the clause ¬*MaxLoad* ∨ `(load ≤ 30)`, and *MaxLoad* could no longer be purely positive.

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

Figure 3: Core LPSAT algorithm (without learning or backjumping). BAD? 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 Σ and simplifying.

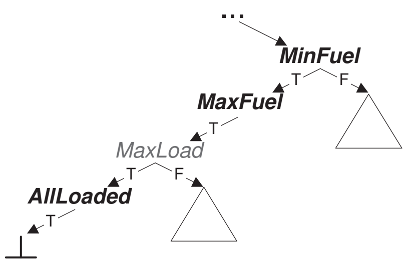

Figure 4: Possible search tree for the constraints from Figure 2. Each node is labeled with the variable set at that node; branchpoints have true (T) and false (F) branches. $\perp$ indicates an inconsistent constraint set. The bold variables are members of the conflict set.

depth-first search of the propositional search space creates a partial assignment to the boolean variables. When the search fails, 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 learn in two ways. First, since making the truth assignments represented in the conflict set leads inevitably to failure, LPSAT can learn a clause disallowing 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 points at which a conflict set variable received its assignment, ignoring any deeper branch points. Figure 4 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

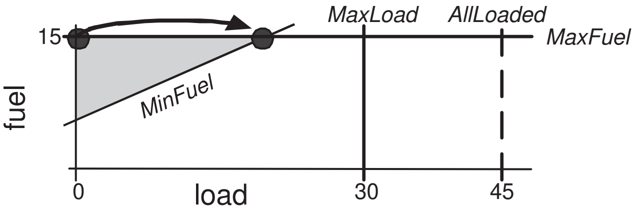

Figure 5: Graphical depiction of the constraints from Figure 2. The shaded area represents solutions to the set of solid-line constraints. The dashed *AllLoaded* constraint causes an inconsistency.

MaxLoad to the branchpoint for *MinFuel*, the deepest member of the conflict set which is a branchpoint.

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. Thus, given a propositional failure LPSAT uses RELSAT's conflict set discovery mechanism unchanged, learning a set based on two of the clauses which led to the contradiction [Bayardo and Schrag, 1997]. For a constraint conflict, however, LPSAT identifies an inconsistent subset of the active constraints, and the propositional triggers for these constraint compose the conflict set. We examine two methods for identifying these inconsistent subsets.

In our first method, called *global conflict set discovery*, LPSAT includes the entire set of active constraints in the conflict set. This mechanism is simple but often suboptimal since a smaller conflict set would provide greater pruning action. Indeed, preliminary experiments showed that — while global conflict set discovery *did* increase solver speed over a solver with no learning or backjumping facility — the conflict sets were on average twice as large as those found for logical conflicts.

In our second method, called *minimal conflict set discovery*, LPSAT identifies a (potentially) much smaller set of constraints which are responsible for the conflict. Specifically, our technique identifies an inconsistent conflict set of which every proper subset is consistent.

Figure 5 illustrates the constraints from the example in Figure 2. 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. Informally, LPSAT finds a minimal conflict set by identifying only those constraints which are, together, in greatest conflict with the new constraint. 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 which 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 5, 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 uses "marker variables" (which CASSOWARY adds to each original constraint) to establish the conflict set. 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.

In Figure 5 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 with even smaller cardinality than the one we find. In general, there may be many minimal conflict sets, and our conflict discovery technique will discovery only one of these, with no guarantee of discovering the global minimum. Some of these sets may prove to have better pruning action, but we know of no way to find the best minimal conflict set efficiently. However, the minimal conflict set *is* at least as good as (and usually better than) any of its supersets.

A brief proof that our technique will return a *minimal* conflict set appears in the longer version of this paper [Wolfman and Weld, 1999].

5 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.

5.1 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[Nagoya,Stockholm]`.

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

Figure 6: 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].

5.2 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 6. In order to make parallel

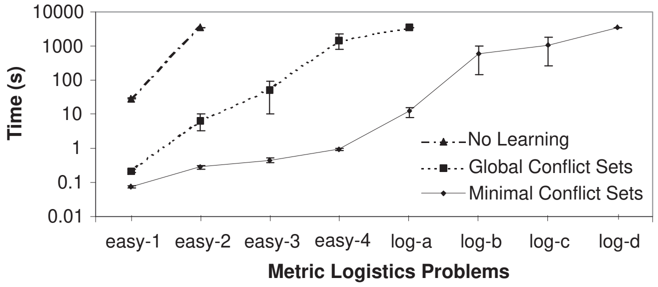

Figure 7: Solution times for three versions of LPSAT in the metric logistics domain. No learning or backjumping is performed in the line marked "No learning." Global conflict sets and minimal conflict sets use progressively better learning algorithms. Note that the final point on each curve reaches the resource cutoff (one hour).

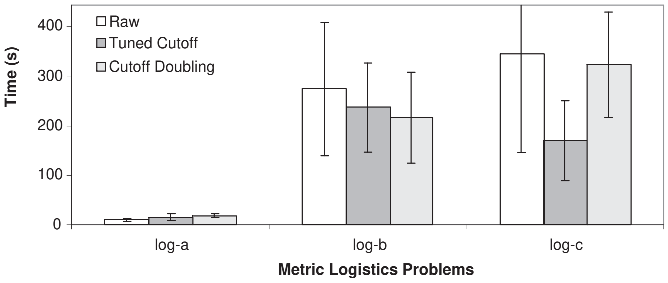

Figure 8: Solution times for two types of random restarts. Tuned cutoff uses raw experimental data to select a constant cutoff. Cutoff doubling starts with a cutoff of one second and doubles it on each run.

actions arbitrarily serializable, we would have to adopt more restrictive exclusivity conditions and a less expressive format for our `test` statements.

6 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 describing the heuristics and optimization we used to enhance LPSAT's performance. 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 *log-c*).

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. *log-a* through *log-d* are the same as the ATT problems except for the addition of fuel. *easy-1* through *easy-4* are simplifications of *log-a* with more elements retained in the higher numbered problems. We report on highly successful experiments with learning and backjumping as well as two other interesting optimizations.

6.1 Learning and Backjumping

The results in Figure 7 demonstrate the improvement in solving times resulting from activating the learning and backjumping facilities which were described in Section 4. Runs were cut off after one hour of solve time (the minimal conflict set technique ran over an hour only on *log-d*). Without learning or backjumping LPSAT quickly exceeds the 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.

6.2 Splitting Heuristic

Line 7 of the LPSAT pseudocode (Figure 3) makes a nondeterministic choice of variable v before the recursive call, and the *splitting heuristic* used to guide this choice can bias performance. We expected the standard RELSAT heuristic to perform poorly (due to a overly strong preference for trigger variables) for two reasons: 1) the trigger itself is an implicit clause which is resolved when a trigger variable is set, and 2) 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, however, we were unable to achieve significant improvement (although *increasing* the preference for trigger variables did slow execution). These results lead us to suspect that either that LCNF problems are generally insensitive to our heuristics or that our compilation of metric planning domains already encodes information about trigger variables in the structure of the problem. Further experiments will decide the issue.

6.3 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 8 shows the results of these experiments. We first ran the algorithm twenty times on each problem to produce the "Raw" entries[4]. Then, we calculated the cutoff time which 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 *log-b* and impressive speedup on *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 8 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 *log-c* five of the twenty "Raw" runs lasted longer than the longest "Cutoff doubling" run.

7 Related Work

Limited space precludes a survey of propositional 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.

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. CHIP [Van Hentenryck, 1989] augments logic programming with the tools to efficiently solve constraint satisfaction problems (*e.g.*, consistency checking), but deals only with variables over finite domains. NUMERICA extends this work by adding a variety of differential equation solvers to the mix [Van Hentenryck, 1997]. Hooker et al. describe a technique for combining linear programming and constraint propagation [Hooker *et al.*, 1999].

BLACKBOX uses a translate/solve/decode scheme for planning and satisfiability [Kautz and Selman, 1998]. ZENO is a causal link temporal planner which handled 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]. SIPE [Wilkins, 1990] and OPLAN [Currie and Tate, 1991] are industrial strength planners which include resource planning capabilities. Two recent systems address the metric planning problem using compilation to integer programming [Kautz and Walser, 1999; Vossen *et al.*, 1999].

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

8 Conclusions and Future Work

LPSAT is a promising new technique that combines the strengths of fast satisfiability methods with an incremental Simplex algorithm to efficiently handle problems involving both propositional and metric reasoning. This paper makes 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 experimented with three optimizations for LPSAT: adapting the splitting heuristic to trigger variables, adding random restarts, and incorporating learning and backjumping. Using *minimal conflict sets* to guide learning and backjumping 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. There are many ways we could improve the compiler: improving its runtime by optimizing exclusion detection, exploring new exclusion encodings, optimizing the number of constraints used for influences, and improving our handling of conditional effects. In addition, 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. We hope to add support for more expressive constraints by adding nonlinear solvers.

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.

[Currie and Tate, 1991] K. Currie and A. Tate. O-plan: the open planning architecture. *Artificial Intelligence*, 52(1):49–86, November 1991.

[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. Reprinted in [?].

[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.

[Nelson and Oppen, 1979] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[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 Hentenryck, 1989] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

[Van Hentenryck, 1997] P. Van Hentenryck. Numerica: A modeling language for global optimization. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, 1997.

[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.

[Wilkins, 1990] D. Wilkins. Can AI planners solve practical problems? *Computational Intelligence*, 6(4):232–246, November 1990.

[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.