Local-search techniques for propositional logic extended with cardinality constraints

Lengning Liu and Mirosław Truszczyński

Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA

Abstract. We study local-search satisfiability solvers for propositional logic extended with cardinality atoms, that is, expressions that provide explicit ways to model constraints on cardinalities of sets. Adding cardinality atoms to the language of propositional logic facilitates modeling search problems and often results in concise encodings. We propose two "native" local-search solvers for theories in the extended language. We also describe techniques to reduce the problem to standard propositional satisfiability and allow us to use off-the-shelf SAT solvers. We study these methods experimentally. Our general finding is that native solvers designed specifically for the extended language perform better than indirect methods relying on SAT solvers.

1 Introduction

We propose and study local-search satisfiability solvers for an extension of propositional logic with explicit means to represent cardinality constraints.

In recent years, propositional logic has been attracting considerable attention as a general-purpose modeling and computing tool, well suited for solving search problems. For instance, to solve a graph k-coloring problem for an undirected graph G, we construct a propositional theory T so that its models encode k-colorings of G and there is a polynomial-time method to reconstruct a k-colorings of G from a model of T. Once we have such a theory T, we apply to it a satisfiability solver, find a model of T and reconstruct from the model the corresponding k-coloring of G.

Instances of many other search problems can be represented in a similar way as propositional theories and this modeling capability of the propositional logic has been known for a long time. However, it has been only recently that we saw a dramatic improvement in the performance of programs to compute models of propositional theories [12, 8, 14, 9, 10, 6]. These new programs can often handle theories consisting of hundreds of thousands, sometimes millions, of clauses. They demonstrate that propositional logic is not only a tool to represent problems but also a viable computational formalism.

The approach we outlined above has its limitations. The repertoire of operators available for building formulas to represent problem constraints is restricted to boolean connectives. Moreover, since satisfiability solvers usually require CNF theories as input, for the most part the only formulas one can use to express

constraints are clauses. One effect of these restrictions is often very large size of CNF theories needed to represent even quite simple constraints and, consequently, poorer effectiveness of satisfiability solvers in computing answers to search problems. Researchers recognized this limitation of propositional logic. They proposed extensions to the basic language with the equivalence operator [7], with cardinality atoms [3, 5] and with pseudo-boolean constraints [2, 13, 1, 4, 11], and developed solvers capable of computing models for theories in the expanded syntax.

In this paper, we focus on an extension of propositional logic with *cardinality atoms*, as described in [5]. Specifically, a cardinality atom is an expression of the form kXm, where k and m are non-negative integers and X is a set of propositional atoms. Cardinality atoms offer a *direct* means to represent cardinality constraints on sets and help construct concise encodings of many search problems. We call this extension of the propositional logic the *propositional logic with cardinality constraints* and denote it by PL^{cc}.

To make the logic PL^{cc} into a computational mechanism, we need programs to compute models of PL^{cc} theories. One possible approach is to *compile cardinality atoms away*, replacing them with equivalent propositional-logic representations. After converting the resulting theories to CNF, we can use any off-the-shelf satisfiability solver to compute models. Another approach is to design solvers specifically tailored to the expanded syntax of the logic PL^{cc}. To the best of our knowledge, the first such solver was proposed in [3]. A more recent solver, *aspps*[1], was described in [5].

These two solvers are *complete* solvers. In this paper, we propose and study *local-search* satisfiability solvers that can handle the extended syntax of the logic PL^{cc}. In our work we built on ideas first used in *WSAT*, one of the most effective local-search satisfiability solvers for propositional logic [12][2]. In particular, as in *WSAT*, we proceed by executing a prespecified number of *tries*. Each try starts with a random truth assignment and consists of a sequence of local modification steps called *flips*. Each flip is determined by an atom selected from an *unsatisfied* clause. We base the choice of an atom on the value of its *break-count* (some measure of how much the corresponding flip increases the degree to which the clauses in the theory are violated). In *WSAT*, the break-count of an atom is the number of clauses that become unsatisfied when the truth value of the atom is flipped. In the presence of cardinality atoms, this simple measure does not lead to satisfactory algorithms and modifications are necessary.

In this paper, we propose two approaches. In the first of them, we change the definition of the *break-count*. To this end, we exploit the fact that cardinality atoms are only high-level shorthands for some special propositional theories and, as we already indicated earlier, can be *compiled* away. Let T be a PL^{cc} theory and let T' be its propositional-logic equivalent. We define the break-count of an atom a in T as the number of clauses *in the compiled theory* T' that become unsatisfied after we flip a. Important thing to note is that we do not need to compute T'

[1] The acronym for answer-set programming with propositional schemata.
[2] In the paper, we write *WSAT* instead of *WALKSAT* to shorten the notation.

explicitly in order to compute the break-count of a. It can be computed directly on the basis of T alone.

Our second approach keeps the concept of the break-count exactly as it is defined in *WSAT* but changes the notion of a flip. This approach applies whenever a PL^{cc} theory T can be separated into two parts T_1 and T_2 so that: (1) T_2 consists of propositional clauses, (2) it is easy to construct random assignments that satisfy T_1, and (3) for every truth assignment satisfying T_1, (modified) flips executed on this assignment result in assignments that also satisfy T_1. In such cases, we can start a try by generating an initial truth assignment to satisfy all clauses in T_1, and then executing a sequence of (modified) flips, choosing atoms for flipping based on the number of clauses in T_2 (which are all standard propositional CNF clauses) that become unsatisfiable after the flip.

In the paper, we develop and implement both ideas. We study experimentally the performance of our algorithms on several search problems: the graph coloring problem, the vertex-cover problem and the open latin-square problem. We compare the performance of our algorithms to that of selected SAT solvers executed on CNF theories obtained from PL^{cc} theories by compiling away cardinality atoms.

2 Logic PL^{cc}

The language of the logic PL^{cc} is determined by the set At of *propositional atoms* and two special symbols $\bot$ and $\top$ that we always interpret as *false* and *true*, respectively. A *cardinality atom* (c-atom, for short) is an expression of the form kXm, where X is a set of propositional atoms, and k and m are non-negative integers. If $X = \{a_1, \ldots, a_n\}$, we will also write $k\{a_1, \ldots, a_n\}m$ to denote a c-atom kXm. One (but not both) of k and m may be missing. Intuitively, a c-atom kXm means: *at least k and no more than m of atoms in X are true*. If k (or m) is missing, the c-atom constrains the number of its propositional atoms that must be true only from above (only from below, respectively).

A *clause* is an expression of the form $\neg\alpha_1 \vee \ldots \vee \neg\alpha_r \vee \beta_1 \vee \ldots \vee \beta_s$, where each α_i, $1 \leq i \leq r$, and each β_j, $1 \leq j \leq s$, is a propositional atom or a c-atom. A *theory* of the logic PL^{cc} is any set of clauses[3].

An *interpretation* is an assignment of truth values $\mathbf{t}$ and $\mathbf{f}$ to atoms in At. An interpretation I *satisfies* an atom a if $I(a) = \mathbf{t}$. An interpretation I satisfies a c-atom $k\{a_1, \ldots, a_n\}m$ if $k \leq |\{i\colon I(a_i) = \mathbf{t}\}| \leq m$.

This notion of satisfiability extends in a standard way to clauses and theories. We will write interchangingly "is a model of" and "satisfies". We will also write $I \models E$, when I is a model of an atom, c-atom, clause or theory E.

We will now illustrate the use of the logic PL^{cc} as a modeling tool by presenting PL^{cc} theories that encode (1) the graph-coloring problem, (2) the graph

[3] It is easy to extend the language of PL^{cc} and introduce arbitrary formulas built of atoms and c-atoms by means of logical connectives. Since clausal theories, as in propositional logic, are most fundamental, we focus on clausal theories only.

vertex-cover problem, and (3) the open latin-square problem. We later use these theories as benchmarks in performance tests.

In the first of these problems we are given a graph G with the set $V = \{1, \ldots, n\}$ of vertices and a set E of edges (unordered pairs of vertices). We are also given a set $C = \{1, \ldots, k\}$ of colors. The objective is to find an assignment of colors to vertices so that for every edge, its vertices get different colors. A PL^{cc} theory representing this problem is built of propositional atoms $c_{i,j}$, where $1 \leq i \leq n$, and $1 \leq j \leq k$. An intended meaning of an atom $c_{i,j}$ is that *vertex* i *gets color* j. We define the theory $col(G, k)$ to consist of the following clauses:

1. $1\{c_{i,1}, \ldots, c_{i,k}\}1$, for every i, $1 \leq i \leq n$. These clauses ensure that every vertex obtains *exactly one* color
2. $\neg c_{p,j} \vee \neg c_{r,j}$, for every edge $\{p, r\} \in E$ and for every color j. These clauses enforce the main colorability constraint.

It is easy to see that models of the theory $col(G, k)$ are indeed in one-to-one correspondence with k-colorings of G.

In a similar way, we construct a theory $vc(G, k)$ that represents the *vertex-cover* problem. Let G be an undirected graph with the set $V = \{1, \ldots, n\}$ of vertices and a set E of edges. Given G and a positive integer k, the objective is to find a set U of no more than k vertices, such that every edge has at least one of its vertices in U (such sets U are *vertex covers*). We build the theory $vc(G, k)$ of atoms in_i, $1 \leq i \leq n$, (intended meaning of in_i: vertex i is in a vertex cover) and define it to consist of the following clauses:

1. $\{in_1, \ldots, in_n\}k$. This clause guarantees that at most k vertices are chosen to a vertex cover
2. $in_p \vee in_r$, for every edge $\{p, r\} \in E$. These clauses enforce the main vertex cover constraint.

Again, it is evident that models of theory $vc(G, k)$ are in one-to-one correspondence with those vertex covers of G that have no more than k elements.

In the open latin-square problem, we are given an integer n and a collection D of triples (i, j, k), where i, j and k are integers from $\{1, \ldots, n\}$. The goal is to find an $n \times n$ array A such that all entries in A are integers from $\{1, \ldots, n\}$, no row and column of A contains two identical integers, and for every $(i, j, k) \in D$, $A(i, j) = k$. In other words, we are looking for a *latin square* of order n that extends the partial assignment specified by D. To represent this problem we construct a PL^{cc} theory $ls(n, D)$ consisting of the following clauses:

1. $a_{i,j,k}$, for every $(i, j, k) \in D$ (to represent the partial assignment D given as input)
2. $1\{a_{i,j,1}, \ldots, a_{i,j,n}\}1$, for every $i, j = 1, \ldots, n$ (to enforce that every entry receives exactly one value)
3. $\{a_{i,1,k}, \ldots, a_{i,n,k}\}1$, for every $i, k = 1, \ldots, n$ (in combination with (2) these clauses enforce that an integer k appears exactly once in a row i)
4. $\{a_{1,j,k}, \ldots, a_{n,j,k}\}1$, for every $j, k = 1, \ldots, n$ (in combination with (2) these clauses enforce that an integer k appears exactly once in a column j).

One can verify that models of the theory $ls(n, D)$ correspond to solutions to the open latin-square problem with input D.

The use of c-atoms in all these three examples results in concise representations of the corresponding problems. Clearly, we could eliminate c-atoms and replace the constraints they represent by equivalent CNF theories. However, the encodings become less direct, less concise and more complex.

3 Using SAT Solvers to Compute Models of PL^{cc} Theories

We will now discuss methods to find models of PL^{cc} theories by means of standard SAT solvers. A key idea is to compile away c-atoms by replacing them with their propositional-logic descriptions. We will propose several ways to do so.

Let us consider a c-atom $C = k\{a_1, ..., a_n\}m$ and let us define a CNF theory C' to consist of the following clauses:

1. $\neg a_{i_1} \vee ... \vee \neg a_{i_{m+1}}$, for any $m+1$ atoms $a_{i_1}, ..., a_{i_{m+1}}$ from $\{a_1, ..., a_n\}$ (there are $\binom{n}{m+1}$ such clauses); and
2. $a_{i_1} \vee ... \vee a_{i_{n-k+1}}$, for any $n-k+1$ atoms $a_{i_1}, ..., a_{i_{n-k+1}}$ in $\{a_1, ..., a_n\}$ (there are $\binom{n}{k-1}$ such clauses).

It is easy to see that the theory C' has the same models as the c-atom C.

Let T be a PL^{cc} theory. We denote by *compile-basic*(T) the CNF theory obtained from T by replacing every c-atom C with the conjunction of clauses in C' and by applying distributivity to transform the resulting theory into the CNF. This approach translates T into a theory in the same language but it is practical only if k and m are small (do not exceed, say 2). Otherwise, the size of the theory *compile-basic*(T) quickly gets too large for SAT solvers to be effective.

Our next method to compile away c-atoms depends on counting. To simplify the presentation, we will describe it in the case of a c-atom of the form kX but it extends easily to the general case. We will assume that $k \geq 1$ (otherwise, kX is true) and $k \leq |X|$ (otherwise kX is false).

Let us consider a PL^{cc} theory T and let us assume that T contains a c-atom of the form $C = k\{a_1, \ldots, a_n\}$. We introduce new propositional atoms: $b_{i,j}$, $i = 0, \ldots, n$; $j = 0, \ldots, k$. The intended role for $b_{i,j}$ is to represent the fact that at least j atoms in $\{a_1, \ldots, a_i\}$ are true. Therefore, we define a theory C' to consist of the following clauses:

1. $b_{0,j} \leftrightarrow \bot$, $j = 1, \ldots, k$,
2. $b_{i,0} \leftrightarrow \top$, $i = 0, \ldots, n$,
3. $b_{i,j} \leftrightarrow b_{i-1,j} \vee (b_{i-1,j-1} \wedge a_i)$, $i = 1, \ldots, n$, $j = 1, \ldots, k$.

Let I be an interpretation such that $I \models C'$. One can verify that $I \models b_{i,j}$ if and only if $I \models j\{a_1, \ldots, a_i\}$. In particular, $I \models b_{n,k}$ if and only if $I \models C$. Thus, if we replace C in T with $b_{n,k}$ and add to T the theory C' the resulting theory has the same models (modulo new atoms) as T. By repeated application

of this procedure, we can eliminate all c-atoms from T. Moreover, if we represent theories C' in CNF, the resulting theory will itself be in CNF. We will denote this CNF theory as *compile-uc*(T), where *uc* stands for *unary* counting. One can show that the size of *compile-uc*(T) is $O(R \times size(T))$, where R is the maximum of all integers appearing in T as lower or upper bounds in c-atoms. It follows that, in general, this translation leads to more concise theories than *compile-basic*. However, it does introduce new atoms.

The idea of counting can be pushed further. Namely, we can design a more concise translation than *compile-uc* by following the idea of counting and by representing numbers in the binary system and by building theories to model binary counting and comparison. For a PL^{cc} theory T, we denote the result of applying this translation method to T by *compile-bc* (*bc* stands for *binary* counting). Due to space limitation we omit the details of this translation. We only note that the size of *compile-bc*(T) is $O(size(T)\log_2(R+1))$, where R is the maximum of all integer bounds of c-atoms appearing in T.

4 Local-search Algorithms for the Logic PL^{cc}

In this section we describe a local-search algorithm *Generic-WSAT*^{cc} designed to test satisfiability of theories in the logic PL^{cc}. It follows a general pattern of *WSAT* [12]. The algorithm executes *Max-Tries* independent *tries*. Each try starts in a randomly generated truth assignment and consists of a sequence of up to *Max-Flips flips*, that is, local changes to the current truth assignment. The algorithm terminates with a truth assignment that is a model of the input theory, or with no output at all (even though the input theory may in fact be satisfiable). We provide a detailed description of the algorithm *Generic-WSAT*^{cc} in Figure 1.

We note that the procedure *Flip* may, in general, depend on the input theory T. It is not the case in *WSAT* and other similar algorithms but it is so in one of the algorithms we propose in the paper. Thus, we include T as one of the arguments of the procedure *Flip*.

We also note that in the algorithm, we use several parameters that, in our implementations, we enter from the command line. They are *Max-Tries*, *Max-Flips* and p. All these parameters affect the performance of the program. We come back to this matter later in Section 5.

To obtain a concrete implementation of the algorithm *Generic-WSAT*^{cc}, we need to define *break-count*(x) and to specify the notion of a *flip*. In this paper we follow two basic directions. In the first of them, we use a simple notion of a flip, that is, we always flip just one atom. We introduce, however, a more complex concept of the break-count, which we call the *virtual break-count*. In the second approach, we use a simple notion of the break-count — the number of clauses that become unsatisfied — but introduce a more complex concept of a flip, which we call the *double-flip*.

To specify our first instantiation of the algorithm *Generic-WSAT*$^{cc}(T)$, we define the break-count of an atom x in T as the number of clauses in the CNF

Figure 1 Algorithm *Generic-WSAT*$^{cc}(T)$

```
INPUT:    T - a PL^cc theory
OUTPUT:   σ - a satisfying assignment of T, or no output
BEGIN
1.    For i ← 1 to Max-Tries, do
2.       σ ← randomly generated truth assignment;
3.       For j ← 1 to Max-Flips, do
4.          If σ ⊨ T then return σ;
5.          C ← randomly selected unsatisfied clause;
6.          For each atom x in C, compute break-count(x);
7.          If any of these atoms has break-count 0 then
8.             randomly choose an atom with break-count 0, call it a;
9.          Else
10.            with probability p, a ← an atom x with minimum break-count(x);
11.            with probability 1 − p, a ← a randomly chosen atom in C;
12.         End If
13.         σ ← Flip(T, σ, a);
14.      End for of j
15.   End for of i
END
```

theory *compile-basic*(T) that become unsatisfied after flipping x. The key idea is to observe that this number can be computed strictly on the basis of T, that is, *without* actually constructing the theory *compile-basic*(T). It is critical since the size of the theory *compile-basic*(T) is in general much larger than the size of T (sometimes even exponentially larger). We refer to this notion of the break-count as the *virtual* break-count as it is defined *not* with respect to an input PL^{cc} theory T but with respect to a "virtual" theory *compile-basic*(T), which we do not explicitly construct.

Further, we define the procedure *Flip*(σ, a) (it does not depend on T hence, we dropped T from the notation) so that, given a truth assignment σ and an atom a, it returns the truth assignment σ' obtained from σ by setting $\sigma'(a)$ to the dual value of $\sigma(a)$ and by keeping all other truth values in σ unchanged (this is the basic notion of the flip that is used in many local-search algorithms, in particular in *WSAT*). We call the resulting version of the the algorithm *Generic-WSAT*$^{cc}(T)$, the *virtual break-count WSAT*^{cc} and denote it by *vb-WSAT*^{cc}.

The second instantiation of the algorithm *Generic-WSAT*^{cc} that we will discuss applies only to PL^{cc} theories of some special syntactic form. A PL^{cc} theory T is *simple*, if $T = T^{cc} \cup T^{cnf}$, where $T^{cc} \cap T^{cnf} = \emptyset$ and

1. T^{cc} consists of *unit* clauses $C_i = k_i X_i m_i$, $1 \leq i \leq p$, such that sets X_i are pairwise disjoint
2. T^{cnf} consists of propositional clauses
3. for every i, $1 \leq i \leq p$, $k_i < |X_i|$ and $m_i > 0$.

Condition (3) is not particularly restrictive. In particular, it excludes c-atoms kXm such that $k > |X|$, which are trivially false and can be simplified away

Figure 2 Algorithm $Flip(T, \sigma, a)$

INPUT: T - a simple PL^{cc} theory ($T = T^{cc} \cup T^{cnf}$)
σ - current truth assignment
a - an atom chosen to flip
OUTPUT: σ - updated σ after a is flipped
BEGIN
1. **If** a occurs in a clause in T^{cc} **and** flipping a will break it **then**
2. pick the best opposite atom, say b, in that clause w.r.t. break-count;
3. $\sigma(b) \leftarrow$ dual of $\sigma(b)$;
4. **End if**
5. $\sigma(a) \leftarrow$ dual of $\sigma(a)$;
6. **return** σ;

END

from the theory, as well as those for which $k = |X|$, which forces all atoms in X to be true and again implies straightforward simplifications. The effect of the restriction $m > 0$ is similar; it eliminates c-atoms with $m = 0$, for which it must be that all atoms in X be false. We note that PL^{cc} theories we proposed as encodings of the graph-coloring and vertex-cover problems are simple; the theory encoding the latin-square problem is not.

In this section, we consider only simple PL^{cc} theories. Let us assume that we designed the procedure $Flip(T, \sigma, a)$ so that it has the following property:

(DF) if a truth assignment σ is a model of T^{cc} then $\sigma' = Flip(T, \sigma, a)$ is also a model of T^{cc}.

Let us consider a try starting with a truth assignment σ that satisfies all clauses in T^{cc}. If our procedure *Flip* satisfies the property (DF), then all truth assignments that we will generate in this try satisfy all clauses in T^{cc}. It follows that the only clauses that can become unsatisfied during the try are the propositional clauses in T^{cnf}. Consequently, in order to compute the break-count of an atom, we only need to consider the CNF theory T^{cnf} and count how many clauses in T^{cnf} become unsatisfiable when we perform a flip.

Since all c-atoms in T^{cc} are pairwise disjoint, it is easy to generate random truth assignments that satisfy all these constraints. Thus, it is easy to generate a random starting truth assignment for a try. Moreover, it is also quite straightforward to design a procedure *Flip* so that it satisfies property (DF). We will outline one such procedure now and provide for it a detailed pseudo-code description.

Let us assume that σ is a truth assignment that satisfies all clauses in T^{cc} and that we selected an atom a as the third argument for the procedure *Flip*. If flipping the value of a does not violate any unit clause in T^{cc}, the procedure $Flip(T, \sigma, a)$ returns the truth assignment obtained from σ by flipping the value of a. Otherwise, since the c-atoms forming the clauses in T^{cc} are pairwise disjoint, there is exactly one clause in T^{cc}, say kXm, that becomes unsatisfied when the value of a is flipped. In this case, clearly, $a \in X$.

We proceed now as follows. We find in X another atom, say b, whose truth value is opposite to that of a, and flip both a and b. That is, $Flip(T, \sigma, a)$ returns the truth assignment obtained from σ by flipping the values assigned to a and b to their duals. Clearly, by performing this *double flip* we maintain the property that all clauses in T^{cc} are still satisfied. Indeed all clauses in T^{cc} other than kXm are not affected by the flips (these clauses contain neither a nor b) and kXm is satisfied because flipping a and b simply switches their truth values and, therefore, does not change the number of atoms in X that are true.

The only question is whether such an atom b can be found. The answer is indeed positive. If $\sigma(a) = \mathbf{t}$ and flipping a breaks clause kXm, we must have that the number of atoms that are true in X is equal to k. Since $|X| > k$, there is an atom in X that is false. The reasoning in the case when $\sigma(a) = \mathbf{f}$ is similar.

A pseudo-code for the procedure is given in Figure 2.

5 Experiments, Results and Discussion

We performed experimental studies of the effectiveness of our local-search algorithms in solving several difficult search problems. For the experiments we selected the graph-coloring problem, the vertex-cover problem and the latin-square problem. We discussed these problems in Section 2 and described PL^{cc} theories that encode them. To build PL^{cc} theories for testing, we randomly generate or otherwise select input instances to these search problems and instantiate the corresponding PL^{cc} encodings. For the graph-coloring and vertex-cover problems we obtain simple PL^{cc} theories and so all methods we discussed apply. The theories we obtain from the latin-square problem are not simple. Consequently, the algorithm *df-WSAT*^{cc} does not apply but all other methods do.

Our primary goal is to demonstrate that our algorithms *vb-WSAT*^{cc} and *df-WSAT*^{cc} can compute models of *large* PL^{cc} theories and, consequently, are effective tools for solving search problems. To this end, we study the performance of these algorithms and compare it to the performance of methods that employ SAT solvers, specifically *WSAT* and *zchaff* [10]. We chose *WSAT* since it is a local-search algorithm, as are *vb-WSAT*^{cc} and *df-WSAT*^{cc}. We chose *zchaff* since it is one of the most advanced *complete* methods. In order to use SAT solvers to compute models of PL^{cc} theories, we executed them on the CNF theories produced by procedures *compile-bc* and *compile-basic* (Section 3). We selected the method *compile-bc* as it results in most concise translations[4]. We selected the method *compile-basic* as it is arguably the most straightforward translation and it does not require auxiliary atoms.

For all local-search algorithms, including *WSAT*, we used the same values of *Max-Tries* and *Max-Flips*: 100 and 100000, respectively. The performance of local-search algorithms depends to a large degree on the on the value of the parameter p (noise). For each method and for each theory, we ran experiments

[4] Our experiments with the translation *compile-uc* show that it performs worse. We believe it is due to larger size of theories it creates. We do not report these results here due to space limitations.

to determine the value of p, for which the performance was the best. All results we report here come from the best runs for each local-search method.

To assess the performance of solvers on families of test theories, we use the following measures.

1. The average running time over all instances in a family
2. The success rate of a method: the ratio of the number of theories in a family, for which the method finds a solution, to the total number of instances in the family for which we were able to find a solution using *any* of the methods we tested (for all methods we set a limit of 2 hours of CPU time/instance).

The success rate is an important measure of the effectiveness of local-search techniques. It is not only important that they run fast but also that they are likely to find models when models exist.

We will now present and discuss the results of our experiments. We start with the coloring problem. We generated for testing five families $C_1, \ldots, C_5$, each consisting of 50 random graphs with 1000 vertices and 3850, 3860, 3870 3880 and 3890 edges, respectively. The problem was to find for these graphs a coloring with 4 colors (each of these graphs has a 4-coloring). We show the results in Table 5. Columns *vb-WSAT*^{cc}, *df-WSAT*^{cc} show the performance results for our local-search algorithms run on PL^{cc} theories encoding the 4-colorability problem on the graphs in the families C_i, $1 \leq i \leq 5$. Columns *WSAT-bc* and *zchaff-bc* show the performance of the algorithms *WSAT* and *zchaff* on CNF theories obtained from the PL^{cc}-theories by the procedure *compile-bc*. Columns *WSAT-basic* and *zchaff-basic* show the performance of the algorithms *WSAT* and *zchaff* on CNF theories produced by the procedure *compile-basic* (since the bounds in c-atoms in the case of 4-coloring are equal to 1, there is no dramatic increase in the size when using the procedure *compile-basic*). The first number in each entry is the average running time in seconds, the second number — the percentage success rate. The results for local-search algorithms were obtained with the value of noise $p = 0.4$ (we found this value to work well for all the methods).

Table 1. Graph-coloring problem

Family	*vb-WSAT*^{cc}	*df-WSAT*^{cc}	*WSAT-bc*	*zchaff-bc*	*WSAT-basic*	*zchaff-basic*
C_1	39/96%	97/100%	27/0%	68/100%	29/100%	91/100%
C_2	40/98%	100/100%	27/0%	142/100%	29/100%	128/100%
C_3	41/100%	103/100%	27/0%	233/100%	30/98%	146/100%
C_4	41/100%	104/98%	28/0%	275/100%	30/96%	216/100%
C_5	42/96%	108/98%	28/0%	478/100%	30/96%	594/100%

In terms of the success rate, our algorithms achieve or come very close to perfect 100%, and are comparable or slightly better than the combination of *compile-basic* and *WSAT*. When comparing the running time, our algorithms are slower but only by a constant factor. The algorithm *vb-WSAT*^{cc} is only about 0.3 times slower and the algorithm *df-WSAT*^{cc} is about 3.5 times slower.

Next, we note that the combination *compile-bc* and *WSAT* does not perform well at all. It fails to find a 4-coloring even for a single graph. We also observe that *zchaff* performs well no matter which technique is used to eliminate c-atoms. It

finds a 4-coloring for every graph that we tested. In terms of the running time there is no significant difference between its performance on theories obtained by *compile-bc* as opposed to *compile-basic*. However, *zchaff* is, in general, slower than *WSAT* and our local-search algorithms *vb-WSAT*^{cc} and *df-WSAT*^{cc}.

Finally, we note that our results suggest that our algorithms are less sensitive to the choice of a value for the noise parameter p. In Table 5 we show the performance results for our two algorithms and for the combination *compile-basic* and *WSAT* on theories obtained from the graphs in the family C_1 and for p assuming values 0.1, 0.2, 0.3 and 0.4.

Table 2. Coloring: sensitivity to the value of p

Noise	*vb-WSAT*^{cc}	*df-WSAT*^{cc}	*WSAT-basic*
$p = 0.1$	16%	100%	18%
$p = 0.2$	98%	100%	90%
$p = 0.3$	100%	100%	98%
$p = 0.4$	96%	100%	100%

We also tested our algorithms on graph-coloring instances that were used in the graph-coloring competition at the CP-2002 conference. We refer to `http://mat.gsia.cmu.edu/COLORING02/` for details. We experimented with 63 instances available there. For each of these graphs, we identified the smallest number of colors that is known to suffice to color it. We then tested whether the algorithms *vb-WSAT*^{cc}, *WSAT* and *zchaff* (the latter two in combination with the procedure *compile-basic* to produce a CNF encoding) can find a coloring using that many colors. We found that the algorithms *df-WSAT*^{cc}, *WSAT* and *zchaff* (the latter two in combination with *compile-basic*) were very effective. Their success rate (the percentage of instances for which these methods could match the best known result) was 62%, 56% and 54%, respectively. In comparison, the best among the algorithms that participated in the competition, the algorithm MZ, has success rate of 40% only and the success rate of other algorithms does not exceed 30%.

For the vertex cover problem we randomly generated 50 graphs with 200 vertices and 400 edges. For $i = 103, \ldots, 107$, we constructed a family VC_i of PL^{cc} theories encoding, for graphs we generated, the problem of finding a vertex cover of cardinality at most i. For this problem, the translation *compile-basic* is not practical as translating just a single c-atom $\{in_1, \ldots, in_{200}\}i$ requires $\binom{200}{i+1}$ clauses and these numbers are astronomically large for $i = 103, \ldots, 107$. The translation *compile-bc* also does not perform well. Neither *WSAT* nor *zchaff* succeed in finding a solution to even a single instance (as always, within 2 hours of CPU time/instance). Thus, for the vertex-cover problem, we developed yet another CNF encoding, which we refer to as *ad-hoc*. This encoding worked well with *WSAT* but not with *zchaff*. We show the results in Table 5. For this problem, the value of noise $p = 0.1$ worked best for all local-search methods.

Our algorithms perform very well. They have the best running time (with *vb-WSAT*^{cc} being somewhat faster than *df-WSAT*^{cc}) and find solutions for all instances for which we were able to find solutions using these and other techniques. In terms of the success rate *WSAT*, when run on *ad-hoc* translations,

Table 3. Vertex-cover problem: graphs with 200 vertices and 400 edges

Family	*vb-WSAT*^{cc}	*df-WSAT*^{cc}	*WSAT-bc*	*zchaff-bc*	*WSAT*-ad-hoc	*zchaff*-ad-hoc
VC_{103}	117/100%	300/100%	11/0%	7200/0%	1696/100%	7200/0%
VC_{104}	86/100%	225/100%	11/0%	7200/0%	1400/100%	7200/0%
VC_{105}	69/100%	178/100%	11/0%	7200/0%	1191/100%	7200/0%
VC_{106}	29/100%	78/100%	11/0%	7200/0%	848/100%	7200/0%
VC_{107}	10/100%	27/100%	11/0%	7200/0%	671/100%	7200/0%

performed as well as our algorithms but was several (7 to 67, depending on the method and family) times slower.

As in the case of graph coloring, our algorithms again were less sensitive to the choice of the noise value p, as shown in Table 5 (the tests were run on the family VC_{103}.

Table 4. Vertex cover: sensitivity to the value of p

Noise	*vb-WSAT*^{cc}	*df-WSAT*^{cc}	*WSAT-ad-hoc*
$p = 0.1$	100%	100%	100%
$p = 0.2$	100%	100%	100%
$p = 0.3$	100%	100%	58%
$p = 0.4$	100%	100%	33%

We also experimented with the vertex-cover problem for graphs of an order of magnitude larger. We randomly generated 50 graphs, each with 2000 vertices and 4000 edges. For these graphs we constructed a family VC_{1035} consisting of 50 PL^{cc} theories, each encoding the problem of finding a vertex-cover of cardinality at most 1035 in the corresponding graph. With graphs of this size, all compilation methods produce large and complex CNF theories on which both *WSAT* and *zchaff* fail to find even a single solution. Due to the use of c-atoms, the PL^{cc} theories are relatively small. Each consists of 2000 atoms and 4001 clauses and has a total of about 10,000 atom occurrences. Our algorithms *vb-WSAT*^{cc} and *df-WSAT*^{cc} run on each of the theories in under an hour and the algorithm *df-WSAT*^{cc} finds a vertex cover of cardinality at most 1035 for 9 of them. The algorithm *vb-WSAT*^{cc} is about two times faster but has worse success rate: finds solutions only in 7 instances.

The last test concerned the latin-square problem. We assumed $n = 30$ and randomly generated 50 instances of the problem, each specifying values for some 10 entries in the array. Out of these instances we constructed a family *LS* of the corresponding PL^{cc} theories. Since these PL^{cc} theories are not simple, we did not test the algorithm *df-WSAT*^{cc} here. The results are shown in Table 5. For the local-search methods, we used the value of noise $p = 0.1$.

Table 5. Open latin-square problem

vb-WSAT^{cc}	*WSAT-bc*	*zchaff-bc*	*WSAT-basic*	*zchaff-basic*
43/100%	0/0%	5/100%	250/84%	637/96%

These results show that our algorithms are faster than the combination of *WSAT* and *compile-basic* (*compile-bc* again does not work well with *WSAT*) and

have a better success rate. The fastest in this case is, however, the combination of *zchaff* and *compile-bc*. The combination of *zchaff* and *compile-basic* works worse and it is also slower than our algorithms.

6 Conclusions

Overall, our local-search algorithms *vb-WSAT*^{cc} and *df-WSAT*^{cc}, designed explicitly for PL^{cc} theories, perform very well.

It is especially true in the presence of cardinality constraints with large bounds where the ability to handle such constraints directly, without the need to encode them as CNF theories, is essential. It makes it possible for our algorithms to handle large instances of search problems that contain such constraints. We considered one problem in this category, the vertex-cover problem, and demonstrated superior performance of our search algorithms over other techniques. For large instances (we considered graphs with 2000 vertices and 4000 edges and searched for vertex covers of cardinality 1035) SAT solvers are rendered ineffective by the size of CNF encodings and their complexity. Even for instances of much smaller size (search for vertex covers of 103-107 elements in graphs with 200 vertices and 400 edges), our algorithms are many times faster and have a better success rate than *WSAT* (*zchaff* is still ineffective).

Also for PL^{cc} theories that contain only c-atoms of the form $1X1$, $X1$ and $1X$, the ability to handle such constraints directly seems to be an advantage and leads to good performance, especially in terms of the success rate. In the graph-coloring and latin-square problems our algorithms consistently had comparable or higher success rate than methods employing SAT solvers. In terms of time our methods are certainly competitive. For the coloring problem, they were slower than the method based on *WSAT* and *compile-basic* but faster than all other methods. For the latin-square problem, they were slower than the combination of *zchaff* and *compile-bc* but again faster than other methods.

Finally, we note that our methods seem to be easier to use and more robust. SAT-based method have a disadvantage that their performance strongly depends on the selection of the method to compile away c-atoms and no method we studied is consistently better than others. The problem of selecting the right way to compile c-atoms away does not appear in the context of our algorithms. Further, the performance of local-search methods, especially the success rate, highly depends on the value of the noise parameter p. Our results show that our algorithms are less sensitive to changes in p than those that employ *WSAT*, which makes the task of selecting the value for p for our algorithms easier.

These results provide further support to a growing trend in satisfiability research to extend the syntax of propositional logic by constructs to model high-level constraints, and to design solvers that can handle this expanded syntax directly. In the expanded syntax, we obtain more concise representations of search problems. Moreover, these representations are more directly aligned with the inherent structure of the problem. Both factors, we believe, will lead to faster, more effective solvers.

In this paper, we focused on the logic PL^{cc}, an extension of propositional logic with c-atoms, that is, direct means to encode cardinality constraints. The specific contribution of the paper are two local-search algorithms *vb-WSAT*^{cc} and *df-WSAT*^{cc}, tailored to the syntax of the the logic PL^{cc}. These algorithms rely on two ideas. The first of them is to regard a PL^{cc} theory as a compact encoding of a CNF theory modeling the same problem. One can now design local-search algorithms so that they work with a PL^{cc} theory but proceed as propositional SAT solvers would when run on the corresponding propositional encoding. We selected the procedure *compile-basic* to establish the correspondence between PL^{cc} theories and CNF encodings, as it does not require any new propositional variables and makes it easy to simulate propositional local-search solvers. We selected a particular propositional local-search method, *WSAT*, one of the best-performing local-search algorithms. Many other choices are possible. Whether they lead to more effective solvers is an open research problem.

The second idea is to change the notion of a flip. We applied it designing the algorithm *df-WSAT*^{cc} for the class of simple PL^{cc} theories. However, this method applies whenever a PL^{cc} theory T can be partitioned into two parts T_1 and T_2 so that (1) it is easy to generate random truth assignments satisfying constraints in T_1, and (2) there is a notion of a flip that preserves satisfaction of constraints in the first part and allows one, in a sequence of such flips, to reach any point in the search space of truth assignments satisfying constraints in T_1. Identifying specific syntactic classes of PL^{cc} theories and the corresponding notions of a flip is also a promising research direction.

In our experiments we designed compilation techniques to allow us to use SAT solvers in searching for models of PL^{cc} theories. In general, approaches that rely on counting do not work well with *WSAT*, as they introduce too much structure into the theory. The translation *compile-basic* is the best match for *WSAT* (whenever it does not lead to astronomically large theories). All methods seem to work well with *zchaff* at least in some of the cases we studied but none worked well for the vertex-cover problem. To design better techniques to eliminate c-atoms and to make the process of selecting an effective translation systematic rather than ad hoc is another interesting research direction.

Our work is related to [13] and [11], which describe local-search solvers for theories in propositional logic extended by pseudo-boolean constraints. However, the classes of formulas accepted by these two solvers and by ours are different. We use cardinality atoms as generalized "atomic" components of clauses while pseudoboolean constraints have to form unit clauses. On the other hand, pseudoboolean constraints are more general than cardinality atoms. At present, we are comparing the performance of all the solvers on the class of theories that are accepted by all solvers (which includes all theories considered here).

Acknowledgments

The authors are grateful to the reviewers for comments and pointers to papers on extensions of propositional logic by pseudoboolean constraints. This research was supported by the National Science Foundation under Grant No. 0097278.

References

1. F.A. Aloul, A. Ramani, I. Markov, and K. Sakallah. Pbs: a backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of the Fifth International Symposium on Theory and Applications of Satisfiability*, pages 346 – 353, 2002.
2. P. Barth. A davis-putnam based elimination algorithm for linear pseudo-boolean optimization. Technical report, Max-Planck-Institut für Informatik, 1995. MPI-I-95-2-003.
3. B. Benhamou, L. Sais, , and P. Siegel. Two proof procedures for a cardinality based language in propositional calculus. In *Proceedings of STACS-94*, pages 71–82. 1994.
4. H.E. Dixon and M.L. Ginsberg. Inference methods for a pseudo-boolean satisfiability solver. In *The 18th National Conference on Artificial Intelligence (AAAI-2002)*, 2002.
5. D. East and M. Truszczyński. Propositional satisfiability in answer-set programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, volume 2174, pages 138–153. Lecture Notes in Artificial Intelligence, Springer Verlag, 2001. Full version submitted for publication (available at `http://xxx.lanl.gov/abs/cs.LO/0211033`).
6. E. Goldberg and Y. Novikov. Berkmin: a fast and robust sat-solver. In *DATE-2002*, pages 142–149. 2002.
7. C.M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *Proccedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, pages 291–296, 2000.
8. C.M. Li and M. Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, 1997.
9. J.P. Marques-Silva and K.A. Sakallah. GRASP: A new search algorithm for satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.
10. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference (DAC)*, 2001.
11. S.D. Prestwich. Randomised backtracking for linear pseudo-boolean constraint problems. In *Proceedings of the 4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems, CPAIOR-02*, pages 7–20, 2002.
12. B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Seattle, USA, 1994.
13. J.P. Walser. Solving linear pseudo-boolean constraints with local search. In *Proceedings of the 11th Conference on Artificial Intelligence, AAAI-97*, pages 269–274. AAAI Press, 1997.
14. H. Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction (CADE-97)*, pages 308–312, 1997. Lecture Notes in Artificial Intelligence, 1104.