Aspps User’s Manual

Deborah East
Department of Computer Science
Southwest Texas State University
San Marcos, TX 78666, USA

Lengning Liu, Stephen Logsdon,
Victor Marek, and Mirosław Truszczyński
Department of Computer Science
University of Kentucky
Lexington, KY 40506-0046, USA

August 29, 2003
Contents

1 Introduction 1

2 Installation 2
   2.1 Obtaining the Apps System .................................. 2
   2.2 System Requirements ........................................... 2
   2.3 Installing Appsgrnd on Unix Systems ......................... 2
   2.4 Installing Apps on Unix Systems ............................. 2
   2.5 Installing WSAT(CC) on Unix Systems ....................... 3

3 Appsgrnd Input 3
   3.1 Data Files ..................................................... 3
   3.2 Program File .................................................. 4

4 Invoking Appsgrnd 6

5 Appsgrnd Output Format 7
   5.1 Atoms .......................................................... 8
   5.2 C-atoms ....................................................... 8
   5.3 Clauses ....................................................... 8
   5.4 Horn Clauses ................................................ 8

6 Utilities 9
   6.1 Pseudo-Boolean Constraint Solvers .......................... 9
      6.1.1 apps2pbe ................................................. 11
      6.1.2 apps2csp ................................................. 11
      6.1.3 apps2cplex .............................................. 11
      6.1.4 pbc2asppks .............................................. 11
   6.2 SAT Solvers ................................................ 11
      6.2.1 apps2cnf-bc ............................................. 12
      6.2.2 apps2cnf-uc ............................................. 12

7 Invoking Apps 13

8 Apps Output 14
   8.1 apps.stat .................................................... 14
   8.2 Output of Answer Sets ....................................... 14

9 Invoking WSAT(CC) 15

10 WSAT(CC) Output 19
   10.1 wsatcc.stat ................................................ 19
   10.2 Output of Answer Sets ................................... 19

11 Examples 19
   11.1 Graph Coloring ............................................ 19
   11.2 N-Queens .................................................. 20

A Apps System Overview 22
   A.1 Processing Programs in PS+ Format .......................... 22
   A.2 Processing Ground Programs in Formats Other than PL^c ... 23
1 Introduction

The \textit{aspps} system is an answer-set programming system based on the extended logic of propositional schemata, PS$^+$ \cite{PSplus1,PSplus2}. The language of this logic includes constant, variable and predicate symbols. Formulas (clauses) are built of atoms ("regular" first-order atoms, existential atoms and cardinality atoms) and are always written as implications, with a conjunction of atoms as the antecedent and the disjunction of atoms as the consequent. A theory in the logic PS$^+$ is a pair \((D,P)\), where \(D\) is a set of ground atoms (only constant symbols as arguments) representing an \textit{instance of a problem} (input data) and \(P\) is a set of PS$^+$ clauses representing a \textit{program} (an \textit{abstraction of the problem}). The meaning of a PS$^+$ theory \(T = (D,P)\) is given by a \textit{family of PS$^+$ models} \cite{PSplus1,PSplus2}.

The logic PS$^+$ serves as a programming language for modeling search problems. To model a search problem \(\Pi\), the programmer (1) designs a schema (defines predicate and constant symbols) to represent problem instances, (2) using elements of the instance-representation schema, and additional predicates and variables, writes a program \(P\) (collection of clauses of the logic PS$^+$) to capture constraints (specifications) of the problem, and (3) represents a particular instance in this schema, for which the problem \(\Pi\) is to be solved, as a set of ground atoms \(D\). All that is done in such a way that models of the theory \((D,P)\) give solutions to the problem \(\Pi\) for the instance \(D\).

The \textit{aspps} system allows one to compute models of PS$^+$ theories. Thus, it can be used to solve search problems that have been encoded as PS$^+$ theories. First, a theory is grounded to a theory in the propositional logic with cardinality constraints, PL$^{cc}$, using the program \textit{asppsgnd}. The theory obtained in this way is an input to a PL$^{cc}$ solver, which computes models of the ground theory and, hence, of the original theory as well. The \textit{aspps} distribution contains two PL$^{cc}$ solvers: \textit{aspps} and \textit{WSAT(C)}. For information on the features and differences of these two solvers, see Sections 7 and 9, respectively. Using several provided utilities, one can also convert ground theories created by \textit{asppsgnd} to other formats for use with other solvers for Boolean propositional satisfiability and pseudo-Boolean propositional satisfiability. Further utilities allow for the conversion from other ground program formats to PL$^{cc}$ format. The use of these utilities is covered in Section 6. For a graphical overview of the \textit{aspps} system, see Appendix A.

As we noted, the language of logic PS$^+$ accepted by \textit{asppsgnd} includes special constructs (atoms) to model disjunctions (existential atoms) and cardinality constraints on sets (cardinality atoms). We will briefly describe \textit{c-atoms} and \textit{e-atoms} in Section 3; however, for more detailed information, we refer to the \textit{Aspps Programmer's Manual} \cite{AsppsManual}.
2 Installation

2.1 Obtaining the Aspps System

Asppsgrnd, aspps, WSAT(CC), and several related utilities can be obtained at http://www.cs.uky.edu/ai/aspps/.

2.2 System Requirements

Asppsgrnd, aspps, and WSAT(CC) work on most Unix-like systems with the gcc compiler available, including Linux (gcc 2.95.3 and 3.2.2), Solaris (gcc 2.95.3), Freebsd 4.7 (gcc 2.95.4), and the Cygwin environment for Windows (gcc 3.2). The utilities require Perl 5 or greater.

2.3 Installing Asppsgrnd on Unix Systems

Installing asppsgrnd is straightforward:

1. Type gunzip -c asppsgrnd.XXX.XX.XX.tar.gz | tar xvf - to extract the source code archive (where XXX.XX.XX is the version-specific date of asppsgrnd).

2. Type cd asppsgrnd.XXX.XX.XX (where, again, XXX.XX.XX is the version-specific date).

3. Type make to compile asppsgrnd.

4. Install the compiled program by placing it in a directory for executable files. For example, type cp asppsgrnd /usr/local/bin/

2.4 Installing Aspps on Unix Systems

The installation process for aspps is similar.

1. Type gunzip -c aspps.XXX.XX.XX.tar.gz | tar xvf - to extract the source code archive (where XXX.XX.XX is the version-specific date of aspps).

2. Type cd aspps.XXX.XX.XX (where, again, XXX.XX.XX is the version-specific date).

3. Type make to compile aspps.

4. Install the compiled program by placing it in a directory for executable files. For example, type cp aspps /usr/local/bin/
2.5 Installing WSAT(CC) on Unix Systems

The installation process for WSAT(CC) is also similar.

1. Type `gunzip -c wsatcc-current.tar.gz | tar xvf` to extract the source code archive.

2. Type `cd wsatcc-x.x-x` (where x.x.x-x is the version of WSAT(CC) being installed).

3. Type `make` to compile wsatcc.

4. Install the compiled program by placing it in a directory for executable files. For example, type `cp wsatcc /usr/local/bin/`

3 Asppsgrnd Input

Proper input for asppsgrnd consists of one or more data files, which contain input data for a problem, and exactly one program file, which describes the problem itself.

3.1 Data Files

Data atoms are defined as predicate symbols with constant arguments. Each atom statement is terminated by a single period, `. '. Comments and blank lines are allowed in the data files. Comments begin with `%' and continue to the end of the line. The following set of atoms is an example of a data file for an instance of graph coloring.

```plaintext
% This is a data file for the graph
% 3-coloring problem.
% The next line defines the four
% vertices of the input graph.

vtx(1). vtx(2). vtx(3). vtx(4).

% The next line specifies the edges of
% the input graph.

edge(1,4). edge(1,2). edge(3,2).

% Finally, the next line defines three
% available colors: r, g, and b.

clr(r). clr(g). clr(b).
```
The syntax for the aspps system also allows data predicates to be defined with ranges or enumerated in a set notation. We could define the \texttt{vtx} unary predicate as follows:

\[ \texttt{vtx[1...4].} \]

In addition, constants may be used to define the data set at grounding. For example, we could define \texttt{clr} as:

\[ \texttt{clr[1..k].} \]

The value of \( k \) is specified on the command line when executing \texttt{asppsgrnd}. The symbol \( k \) will be replaced by the value given on the command line anywhere in the data files and the program file. For information on how to specify command line constants, see Section 4.

For further details on the format of \texttt{asppsgrnd} data files, see the \textit{Aspps Programmer’s Manual} [3].

### 3.2 Program File

The program file (only one allowed) includes a preamble, where \textit{program predicates} are defined and \textit{variables} are declared. The preamble in the program file restricts the way program predicates are grounded. Program predicate definitions consist of a line:

\[ \texttt{pred name(type}_1,\ldots,\texttt{type}_n).} \]

where \texttt{pred} is a keyword, \textit{name} is the predicate name, and each \texttt{type}_i specifies the accepted type of argument.

Variables are declared as follows:

\[ \texttt{var type varname}_1,\ldots,\texttt{varname}_n]. \]

where \texttt{var} is a keyword, \textit{type} is the type of variable, and each \textit{varname}_i is a variable name. Data predicates are used to define the data types mentioned in the preamble.

The following lines form an example of the preamble for the graph-coloring program.

\[ \texttt{pred color(vtx,clr).} \]
\[ \texttt{var vtx X,Y.} \]
\[ \texttt{var clr K,C.} \]

The program predicate \texttt{color} is a binary predicate. After grounding, its first argument must be a constant from the extension of the data predicate \texttt{vtx} as defined in the data file (in our example, the set \{1, 2, 3, 4\}). Its second argument, after grounding, must be a constant from the extension of the data
predicate \( c1r \) \((\{r, g, b\} \text{ in our example})\). Only unary data predicates can be used to define the types of arguments of program predicates.

\( k \)-ary data predicates can be used in predicate definitions to restrict their extensions. When used in this manner, the data predicates and program predicates must have the same arity. An example (not related to graph coloring) is:

\[
\text{pred } h c(vtx, vtx): \text{edge}.
\]

This statement restricts the extension of the program predicate \( hc \) to a subset of the extension of the data predicate \( \text{edge} \).

The preamble also declares types of variable symbols used in the program. Variables are declared by means of unary data predicates (examples are given above). The declaration of variables allows for more efficient grounding and further error checking.

The preamble is followed by clauses describing constraints of the problem. An example of a program for graph coloring follows.

\[
\text{color}(X, r) \mid \text{color}(X, g) \mid \text{color}(X, b).
\]

\[
\text{color}(X, K), \text{color}(X, C), K \neq C \rightarrow.
\]

\[
\text{color}(X, K), \text{color}(Y, K), \text{edge}(X, Y) \rightarrow.
\]

The first clause ensures that each vertex is assigned at least one color. The second clause ensures that a contradiction is reached if two different colors are assigned to a vertex. The last clause, likewise, prohibits assigning the same color to vertices connected by an edge. As demonstrated by the heads of the last two clauses, the empty string denotes contradiction.

In some cases, the consequent of a clause must be a disjunction of a set of atoms that depends on a particular data instance. To build such disjunctions, we introduced in the language of the logic \( PS^+ \) the notion of an existential atom, or \( e \)-atom. An example of an \( e \)-atom (in the context of our graph-coloring setting) is

\[
\text{color}(X, \_).
\]

It stands for the disjunction of all atoms of the form

\[
\text{color}(X, c)
\]

where \( c \) is a constant from the extension of the data predicate \( c1r \). Since the extension of \( c1r \) in our graph-coloring example is the set \( \{r, g, b\} \), the \( e \)-atom

\[
\text{color}(X, \_).
\]

represents the disjunction

\[
\text{color}(X, r) \mid \text{color}(X, g) \mid \text{color}(X, b).
\]
The use of this special construct for e-atoms allows us to rewrite the first clause of the graph-coloring example as:

\[ \text{color}(X, \_). \]

Another powerful modeling concept in the language of logic PS\(^+\) is that of a \textit{cardinality atom}, or \textit{c-atom}. An example of a cardinality atom is

\[ k\{\text{color}(X, \_)}\}m. \]

The expression within the braces has the same meaning as the e-atom described above. The meaning of the atom

\[ k\{\text{color}(X, \_)}\}m. \]

is: for every \( X \), at least \( k \) and no more than \( m \) atoms of the form

\[ \text{color}(X, c) \]

are true in any answer set. (A more general syntax for c-atoms is available. For details, see the \textit{Aspps Programmer’s Manual} [3].) Using the concept of a cardinality atom, we can replace both the first and second clause of our coloring example with a single clause:

\[ 1\{\text{color}(X, \_)}\}1. \]

In addition to the program and data predicates, the \textit{aspps} implementation includes predefined predicates and function symbols such as comparison operators \( ==, !=, <=, >=, <, > \), arithmetic operators \( +, -, \ast, / \), and functions \texttt{abs} (absolute value), \texttt{mod}, \texttt{max}, and \texttt{min}. We assign to these symbols their standard interpretation. We emphasize that the domains are restricted only to those constants that appear in a theory.

For further details on the format of \textit{asppsgrnd} program files, see the \textit{Aspps Programmer’s Manual} [3].

## 4 Invoking \texttt{asppsgrnd}

The grounding of logic PS\(^+\) programs is performed by the module \textit{asppsgrnd}. The required input to execute \textit{asppsgrnd} is a single program file, one or more data files, and optional constants specified at the command line. If no errors are found while reading the files and during grounding, a machine-readable ground program is generated and printed to the standard output (unless the \texttt{-o} command line option is used). The format of the \textit{asppsgrnd} output is covered in Section 5.

\textit{Asppsgrnd} has the following command line options:

\texttt{asppsgrnd -r rfile -d dfile1 [dfile2 ... dfile\textsubscript{m}] [-c c\textsubscript{1}=v\textsubscript{1} [c\textsubscript{2}=v\textsubscript{2} ... c\textsubscript{n}=v\textsubscript{n}] [-o [theoryfile]] [-C]}

6
Required Arguments:

- `r file`

  `file` is the file describing the problem (the program file). There must be exactly one program file.

- `-d file1 [file2 ... filem]`

  Each `filei` is a data file containing data that will be used to instantiate the theory.

Optional Arguments:

- `-c c1=v1 [c2=v2 ... cn=vn]`

  This option allows the use of constants in both the data and program files. When `ci` is found while reading input files, it is replaced by `vi`. `vi` can be any string that is valid for the data type. If `ci` is to be used in a range specification, then `vi` must be an integer.

- `-o [theoryfile]`

  If `theoryfile` is specified, `asppsgrnd` will save its output to that filename. Otherwise, the name of the output file is a catenation of the constants and the program and data file names, with the extension `.aspps` (or `.cnf`, if the `-C` flag is used).

- `-C`

  This option changes `asppsgrnd`'s output to DIMACS CNF format [5]. It can only be used if no cardinality atoms are present in the program file.

The ordering of the command line options is unimportant, provided the data files and constants are enumerated together in their respective lists.

Example Usage of `Asppsgrnd`:

```
asppsgrnd -r program_file -d data_file -c n=8 -o theory_file.aspps
```

5 \textit{Asppsgrnd} Output Format

The ground programs generated by `asppsgrnd` are similar to the DIMACS CNF format.

- All lines are terminated by a single linefeed character, `\n`.
- Every line that starts with the letter `c` is a comment and is ignored.
- All other lines are the clauses of the ground program.
- Atoms are represented by positive integers.
However, unlike the DIMACS CNF format:

- Each file has a header line of the form:
  
  \[ p \text{ num of atoms} \text{ num of clauses} \]
  
- Clauses are terminated by the end of the line (\n), not with a ‘\n’.

- C-atoms and Horn clauses are preserved and represented using special notation.

5.1 Atoms

We use positive integers from the set \{ 1, 2, ..., \<number of atoms> \} to represent ground atoms. The mapping between a ground atom and its corresponding integer is explicitly listed by the comment lines appearing at the end of each file.

Comments which start with the number ‘0’ represent atoms which were computed during grounding. These atoms are always true and are therefore present in all answer sets. Atoms which are always false do not appear at all in the ground programs.

5.2 C-atoms

A c-atom \{ a_1, a_2, ..., a_n \}m is represented after grounding as:

\[
\{ k \equiv \text{int}_{a_1}, \ldots, \text{int}_{a_n} \}
\]

where \text{int}_{a_i}’s are integer representations of ground atoms in the c-atom. All integers are separated by space characters.

5.3 Clauses

A clause \( A_1, \ldots, A_m \rightarrow B_1 | \ldots | B_n \) is expressed after grounding as:

\[
r(A_1) \ldots r(A_m), r(B_1) \ldots r(B_n)
\]

where \( r(A) = \text{int}_a \) if \( A = a \), for some ground atom \( a \), and \( r(A) = \{ k \equiv \text{int}_{a_1}, \ldots, \text{int}_{a_n} \} \), if \( A = \{ a_1, a_2, \ldots, a_n \}m \) is a cardinality atom (as before, we write \text{int}_a to denote the integer representing a ground atom \( a \)). All integers are separated by space characters, and the body and the head are separated by a comma.

5.4 Horn Clauses

A Horn clause of the form \( h : B_1, \ldots, B_m \) is represented after grounding as:

\[
r(B_1) \ldots r(B_m) : \text{int}_h
\]

where \( r(B)s \) are defined as above, All integers are separated by space characters, and the body and head are separated by a colon character.
6 Utilities

One can use aspsoctd with solvers other than aspps and WSAT(C). As mentioned in Section 4, aspsoctd can also generate output in DIMACS CNF format. Several scripts are also available to convert ground programs in PLcc format to other formats.

6.1 Pseudo-Boolean Constraint Solvers

A pseudo-Boolean constraint is a linear inequality with integer coefficients. The variables in the pseudo-Boolean constraint can take values of either 0 (false) or 1 (true). For more information about pseudo-Boolean constraints, please refer to the website: http://www.cirr.uoregon.edu/PBLIB/.

There are solvers that are designed to deal with pseudo-Boolean constraints, including PBS [6], WSAT(OIP) [7] and Saturn [8]. Moreover, any integer programming software such as CPLEX [9] that is able to deal with linear inequalities is capable of solving propositional satisfiability problems extended with pseudo-Boolean constraints.

There is a natural translation from cardinality atoms to pseudo-Boolean constraints because the cardinality atom in the aspps system is indeed a special type of a pseudo-Boolean constraint, where all the coefficients in the pseudo-Boolean constraint are 1. Thus, a ground PLcc program can be converted into such formats that are suitable for pseudo-Boolean solvers. The only major difference between the use of a cardinality atom and the use of a pseudo-Boolean constraint is that we allow multiple cardinality atoms in one rule while each pseudo-Boolean constraint constitutes a single rule.

For example, the following ground aspps rule is allowed:

\[ v_1, k_1 \{ v_1 \ v_2 \ldots \ v_{n_1} \} m_1 \rightarrow k_2 \{ v_1 \ v_2 \ldots \ v_{n_2} \} m_2 \lor v_3. \]

In order to model the rule in terms of pseudo-Boolean constraints, we need to introduce new propositional variables to represent each cardinality atom in the rule. Namely, we propose the following translation:

\[ v_1, \ c_1 \rightarrow c_2 \lor v_3. \]

where

\[ c_1 \leftrightarrow k_1 \{ v_1 \ v_2 \ldots \ v_{n_1} \} m_1 \]

and

\[ c_2 \leftrightarrow k_2 \{ v_1 \ v_2 \ldots \ v_{n_2} \} m_2. \]

The next step is to define the equivalence relationship between \( c_1 \) and \( k_1 \{ v_1 \ v_2 \ldots \ v_{n_1} \} m_1 \) as well as \( c_2 \) and \( k_2 \{ v_1 \ v_2 \ldots \ v_{n_2} \} m_2 \) in terms of pseudo-Boolean constraints. Here is one implementation:

1. For each cardinality atom of the following form:
we divide it into two cardinality atoms:
\[ k \{ v_1, v_2, \ldots, v_n \} \] \( m \)

Now we use yet another two propositional variables \( z_1 \) and \( z_2 \) as follows:
\[ z_1 \iff k \{ v_1, v_2, \ldots, v_n \} \]
and
\[ z_2 \iff \{ v_1, v_2, \ldots, v_n \} \] \( m \).
Thus the original cardinality atom is equivalent to the conjunction of the two propositional variables:
\[ k \{ v_1, v_2, \ldots, v_n \} \] \( m \) \( \iff \) \( z_1 \land z_2 \).
Then we have
\[ c \iff z_1 \land z_2 \]
where \( c \) is the extra propositional variable we used to replace the cardinality atom in the original rule.

2. Now let us define \( z_1 \) as follows:
\[ z_1 \iff k \{ v_1, v_2, \ldots, v_n \} \iff \]
\[ z_1 \Rightarrow k \{ v_1, v_2, \ldots, v_n \} \text{ and } k \{ v_1, v_2, \ldots, v_n \} \Rightarrow z_1 \iff \]
\[-kz_1 + v_1 + \ldots + v_n >= 0 \text{ and } -nz_1 + v_1 + \ldots + v_n <= (k-1)\]

3. Similarly, we define \( z_2 \) as follows:
\[ z_2 \iff \{ v_1, v_2, \ldots, v_n \} \] \( m \iff \]
\[ z_2 \Rightarrow \{ v_1, v_2, \ldots, v_n \} \] \( m \) \( \text{ and } v_1, v_2, \ldots, v_n \) \( m \Rightarrow z_2 \iff \]
\[ nz_2 + v_1 + \ldots + v_n <= (n+m) \text{ and } (m+1)z_2 + v_1 + \ldots + v_n >= (m+1)\]

From the procedure given above, we can define the equivalence relationship between a propositional variable \( c \) and the cardinality atom \( k \{ v_1, \ldots, v_n \} \) \( m \) by the following propositional clauses and pseudo-Boolean constraints:
\[ c \Rightarrow z_1 \land z_2 \]
\[ z_1 \Rightarrow c \]
\[ z_2 \Rightarrow c \]
\[ -kz_1 + v_1 + \ldots + v_n >= 0 \]
\[ -nz_1 + v_1 + \ldots + v_n <= (k-1) \]
\[ nz_2 + v_1 + \ldots + v_n <= (n+m) \]
\[ (m+1)z_2 + v_1 + \ldots + v_n >= (m+1) \]
In some special cases, for example when the cardinality atom has only one bound or the cardinality atom appears as a unit rule, we can simplify the translation. The details of the simplification are trivial and are left to the reader.

We provide the following scripts to automate the translation procedure:

6.1.1 aspps2pbs

This script translates a ground program into a propositional theory with pseudo-Boolean constraints in PBS format. When it is run, two files are created. One file has the extension .cnf and contains a theory in DIMACS CNF format. The other file is given the extension .cnf.pb and contains all the pseudo-Boolean constraints in PBS format.

6.1.2 aspps2oip

This script translates a ground program into a format that is used by the WSAT(0IP) solver. When run, it creates one file with the extension .oip.

6.1.3 aspps2cplex

This script translates a ground program into an integer programming theory in CPLEX format. It will create a file with the extension .cplex.

All three of the scripts above have the same command line usage:

```
script_name <ground PLcc program file>
```

6.1.4 pbs2aspps

This script translates a propositional theory with pseudo-Boolean constraints in PBS format into a PLcc ground program. The restrictions on the input theory are: (1) pseudo-boolean constraints must have all coefficients equal to 1, and (2) there must not be any optimization constraints.

Usage:

```
pbs2aspps <CNF theory file>
```

6.2 SAT Solvers

Cardinality atoms in ground PLcc programs can also be converted into pure propositional clauses. We can compile away cardinality atoms of a given ground PLcc theory and create a logically equivalent theory in conjunctive normal form. Then, researchers can apply their favorite SAT solvers on the equivalent theory to compute models.

1. Binary-counting:

The cardinality atom is a shortcut for us to model the cardinality constraints on a set of atoms. The idea of binary-counting is to use extra
propositional variables to construct a binary string that counts the number of atoms that obtain truth value TRUE in the given set. In order to build such a binary string, we need $O(\log(k))$ new propositional variables, where $k$ is the larger bound in the given cardinality atom.

Extra propositional clauses are needed so that the extra propositional variables can represent the number. Yet another collection of clauses is used to implement the comparison between two binary numbers.

For more details, please see the paper *Local-search Techniques for Propositional Logic with Cardinality Constraints* [4].

2. Unary-counting:

Similar to binary-counting, unary-counting also counts the number of atoms in the given set that currently obtain the truth value TRUE. Instead of using a binary number, unary-counting uses unary numbers, which leads to a simple encoding but with more extra propositional variables and clauses. The idea is to represent each cardinality atom in a recursive fashion. For example:

$$
k \{ v_1 \ldots v_n \} \leftrightarrow ( k \{ v_1 \ldots v_{n-1} \} \land v_n ) \lor 
( k-1 \{ v_1 \ldots v_{n-1} \} \land v_n )
$$

Then, we replace each cardinality atom in the recursion by an extra propositional variable. The base cases for the recursion are trivial. We refer to the paper *Local-search Techniques for Propositional Logic with Cardinality Constraints* [4] for more details.

We provide the following two translation scripts:

6.2.1 aspps2cnf-bc

This script compiles away c-atoms in a ground program file and generates a propositional theory in DIMACS CNF format using the binary-counting technique. This binary-counting version of aspps2cnf results in smaller theories than the unary-counting version below.

6.2.2 aspps2cnf-uc

This script compiles away c-atoms in a ground program file and generates a propositional theory in DIMACS CNF format using the unary-counting technique.

The two scripts above have the same command line usage:

```
script_name <ground PLcc program file> [-r] [-a]
```

Required Arguments:

```
<ground PLcc program file>
```
This is the name of the $\text{PL}^{cc}$ program file to convert.

Optional Arguments:
- `-r`
  Use a random ordering of atoms in c-atoms.
- `-a`
  Output the result in $\text{PL}^{cc}$ format.

By default, a new file that uses the name of `<ground aspps program file>` with the new suffix `.cnf` will be created. The resulting file is in DIMACS CNF format. If the `-r` flag is specified, a random ordering of the set of atoms in the cardinality atom will be used; otherwise, the order in which the atoms occur will be used. If the `-a` flag is specified, the resulting file will be in $\text{PL}^{cc}$ format (but without c-atoms).

7 Invoking Aspps

The solver, aspps, is used to compute models of the ground $\text{PL}^{cc}$ theory produced by asppsgrnd. The name of the file containing the theory is input on the command line. After executing the aspps program, a file named aspps.stat is created or appended with statistics concerning this execution of aspps. Depending on which command line options were used, aspps may generate additional output. See Section 8 for further information on the output of aspps, including the format of aspps.stat.

Aspps has the following command line options:

```
  aspps -f filename [-A] [-P] [-C [w]] [-L [w]] [-S name, [name2 ...
        name_n] ]
```

Required Arguments:
- `-f filename`
  Where `filename` is the name of the file containing a ground theory in $\text{PL}^{cc}$ format.

Optional Arguments:
- `-A`
  Prints atoms that are true in the computed model, in readable form.
- `-C [w]`
  Counts the number of solutions. If the `-C` flag is not used, aspps stops after the first solution is found or after the whole search space is exhausted, whichever comes first. If the flag is used but `w` is not specified, aspps finds all solutions. Finally, if the flag is used and `w` is specified (where `w` is a positive integer), aspps stops after finding `w` solutions or exhausting the whole search space, whichever comes first.
-L [z]

Enables look-ahead (disabled by default). If z is specified, aspps
will look ahead z atoms. If z is omitted, the default value of 5 will
be used.

-S name1 [name2 ... name_n]

Shows positive atoms built of predicates name_i.

Example Usage of Aspps:

aspps -f theory_file.aspps -A -C

8 Aspps Output

8.1 aspps.stat

When aspps is run, it creates (or updates) a file named aspps.stat. This file
contains statistics about the last run of aspps. Its fields are, in order:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>The number of variables in the input program.</td>
</tr>
<tr>
<td>Clause</td>
<td>The number of clauses in the input program.</td>
</tr>
<tr>
<td>Sat</td>
<td>The status of the theory: sat or unsat, respectively</td>
</tr>
<tr>
<td>CPU Time</td>
<td>The amount of CPU time used, in seconds.</td>
</tr>
<tr>
<td>Branch</td>
<td>The number of branch points.</td>
</tr>
<tr>
<td>Bcktrck</td>
<td>The number of times aspps backtracked.</td>
</tr>
<tr>
<td>Count</td>
<td>The number of solutions found.</td>
</tr>
<tr>
<td>Filename</td>
<td>The filename of the input program.</td>
</tr>
</tbody>
</table>

The fields are separated by one or more spaces, and each run of aspps is
represented by a different line, delimited by the linefeed ('\n') character.

The graph-coloring example from Section 3 will cause aspps to generate lines
similar to the following:

<table>
<thead>
<tr>
<th>Var</th>
<th>Clause</th>
<th>Sat</th>
<th>CPU Time</th>
<th>Branch</th>
<th>Bcktrck</th>
<th>Count</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>13</td>
<td>sat</td>
<td>0</td>
<td>22</td>
<td>24</td>
<td>24</td>
<td>gc.aspps</td>
</tr>
</tbody>
</table>

The line containing column labels is written when the aspps.stat file is first
created. Subsequent executions of aspps append the file only with statistics like
those in the second line.

8.2 Output of Answer Sets

Three aspps parameters affect its output: -A, -C, and -S. When the -A command
line option is used, aspps prints to the standard output atoms that are true in
the computed model. This output is in human-readable form. Atoms which
were determined during grounding to always be true are listed once, at the
beginning. Next, the true atoms from answer sets are listed. The number of answer sets printed is controlled by the -C command line flag, as discussed in Section 7. Any atoms which were determined during grounding to be false are omitted from the output of aspps. If the -S flag is used, aspps will display all positive atoms constructed from the specified predicate (or predicates).

For example, when a solution is found for the graph-coloring example using the command aspps -f gc.aspps -A (where gc.aspps is the name of the filename containing the ground PL^cc program), aspps will output the following:

\texttt{aspps.2003.01.09}

\textbf{Atoms determined during grounding:}

\texttt{Answer-set 1:}
- color(1,r)
- color(2,g)
- color(3,r)
- color(4,g)

9 Invoking \textit{WSAT(CC)}

\textit{WSAT(CC)} is yet another solver for computing models of ground PL^cc theories. It uses a local search technique that was first introduced in Walksat [10]. Because of the existence of cardinality atoms in ground PL^cc theories, we designed \textit{WSAT(CC)} so that it can deal with cardinality atoms directly. Namely, we provide users with three different methods for dealing with cardinality atoms:

1. Virtual breakcount method [4]:
   By default, \textit{WSAT(CC)} uses the concept of breakcount that was introduced in Walksat. When cardinality atoms are present in the theory, we compute the breakcount based on a virtually compiled version of the theory, that is, a propositional theory equivalent to the original (input) one.

2. Double flip method [4]:
   When -DF or --double-flip is specified at the command line and the given PS^+ theory is simple\footnote{A simple PS^+ theory satisfies the following conditions: (a) all the cardinality atoms appear in unit clauses, and (b) all the sets of atoms in the cardinality atoms are pairwise disjoint.}, \textit{WSAT(CC)} will generate only those truth assignments that satisfy all cardinality clauses. The breakcount computation does not involve the virtual theory we mentioned above.

3. Permutation flip method [4]:
   When -P or --permutation-flip is specified and the given PS^+ theory uses cardinality atoms only to define a permutation, \textit{WSAT(CC)} will use the same idea mentioned in the double flip method.
There are some variants in both the double flip method and the permutation flip method. A detailed description of those flags can be found below.

For more information, please refer to the paper *Local-search techniques for propositional logic with cardinality constraints* [4].

**WSAT(CC)** has the following command line options:

```
wsatcc -f filename [OPTIONS]
```

where `filename` is the name of the file containing the ground PL\(^c\) theory and `OPTIONS` may include the following flags:

**Input Format Control:**

- `-D`, `--dimacs`
  
  Indicates that the input file is in the DIMACS CNF format.

- `-DT`, `--data <file name>`
  
  Forces assignments specified in data file `<file name>` to be used at the beginning of each try.

**Computation Control:**

- `-C nn`, `--combination-computation nn`
  
  Specify the method to approximate \(q^n\).

  - `nn = 1` Use linear approximation.
  - `nn = 2` Use quadratic approximation (default).
  - `nn = 3` Use exact computation (often inaccurate due to frequent overflows).

- `-DBF [nn [mm]]`, `--double-flip [nn [mm]]`
  
  Requires that the input theory be simple ((a) all the cardinality atoms appear in unit clauses, and (b) all the sets of atoms in the cardinality atoms are pair-wise disjoint). In each local-search step performs double flip. With the double-flip option, **WSAT(CC)** generates only those truth assignments that satisfy all the cardinality atoms (since all those cardinality atoms appear in unit clauses and have disjoint sets of atoms, those clauses are satisfiable). The double-flip procedure checks whether flipping a selected atom falsifies any cardinality atom (there can be at most one cardinality atom that can get falsified by changing the truth value of a single regular atom). If so, **WSAT(CC)** performs another “single-atom” flip to “re-satisfy” cardinality atom that got unsatisfied. If not, the second flip is performed with probability `nn : mm` (default is 0 : 100).

- `-DBFW [nn [mm]]`, `--double-flip-walk [nn [mm]]`
  
  This option is similar to the `-DBF` option. The only difference is that when the atom to be flipped is chosen by the random walk procedure (and not by the greedy procedure), **WSAT(CC)** will not attempt to fix a cardinality atom that may get unsatisfied after the flip.
-P, --permutation-flip

Requires that the input theory satisfies the following conditions:

1. all cardinality atoms appear in unit clauses
2. the upper and lower bounds of all the cardinality atoms are 1
3. all cardinality atoms can be partitioned into two disjoint groups
   so that: (a) the sizes of the two groups are the same, (b) in each
   partition, all the sets of atoms of the cardinality atoms are
   pairwise disjoint, (c) for any two cardinality atoms that belong
   to different partitions, they share exactly one atom, and (d)
   every atom appearing in a cardinality atom appears in exactly
   two cardinality atoms, one from each group.

These conditions are satisfied if and only if there is a positive
integer n such that atoms appearing in cardinality atoms can be
enumerated \( a_{ij}, i = 1, \ldots, n, j = 1, \ldots, n \), and cardinality
atoms

\[ \{a_{i1}, \ldots, a_{in}\}, i = 1, \ldots, n, \{a_{j1}, \ldots, a_{jn}\}, j = 1, \ldots, n, \]

are the only cardinality atoms in the theory. Models satisfying these
constraints are in one-to-one correspondence with permutations of
\( \{1, \ldots, n\} \). Permutation flip is a collection of four single atom flips
that correspond to the operation of transposition. (For selected i
and j, a transposition swaps values assigned to i and j. For in-
stance, \( (1, 3, 2, 4) \Rightarrow (1, 4, 2, 3) \) is a result of transposition applied
with respect to positions 2 and 4.)

-SW, --single-weight

When double flip or permutation flip is used, each local-search step
may require 2 or 4 single-atom flips. By default, WSAT(CC) com-
putes the sum of the breakcounts of the 2 or 4 atoms to be flipped
as the weight and chooses atoms for flipping based on that weight.
One can use option -SW to force WSAT(CC) to base the selection of
atoms to flip based on individual breakcounts.

-c, --cutoff nn

The cutoff (or the maximum number of flips) of each try. The default
is 100000.

-t, --tries nn

The total number of tries. The default is 10.

-F, --stop-on-first

Stop right after finding the first solution.

-s, --seed nn
The seed of the random number generator.

-N, --noise nn [mm]
The noise ratio (default is 10 : 100). It determines the likelihood with which a random selection procedure is chosen (rather than a greedy one) for selecting an atom to flip.

-T, --true nn [mm]
The probability (nn : mm) of initializing an atom to TRUE (default is 50 : 100).

-V, --verify-solution
Perform solution verification when solutions are found.

Output Control:
-A, --show-all
Output each solution by printing all positive atoms in the solution.

-S, --show name
Output each solution by printing only positive atoms built of the predicate name.

-E, --debug
Turn on debug mode, by which more info will be dumped.

-q, --QUIET
Turn on quiet mode. No statistical info will be dumped.

-q, --no-process-bar
Turn on the mode in which no process bar will be shown. This is useful when one wants to redirect the output to a file and wants to skip all the messy characters used to print the process bar.

-pta, --plot-true-atom-number nn
Generate GnuPlot files for the number of true atoms after nn flips.

-ptt, --plot-touched-times nn
Generate GnuPlot files for the clauses that are touched at least nn times.

Utilities:
-h, --help
Print the help message.

Example Usage of WSAT(CC):
wsatcc -f theory_file.aspps -A
10 \textit{WSAT(CC)} Output

10.1 \texttt{wsatcc.stat}

When \textit{WSAT(CC)} is run, it creates (or updates) a file named \texttt{wsatcc.stat}. This file contains statistics about the last run of \textit{WSAT(CC)}. It has the following fields:

- \texttt{Var} The number of variables in the input program.
- \texttt{Clause} The number of clauses in the input program.
- \texttt{Seed} The seed of the random number generator.
- \texttt{CPU Time} The amount of CPU time used, in seconds.
- \texttt{Tries} The number of tries.
- \texttt{Cutoff} The cutoff.
- \texttt{ASFMUC} If solutions were found, this field contains the average number of flips used in the success tries; otherwise, it is the average number of unsatisfied clauses after each try.
- \texttt{SR} The success rate.
- \texttt{Filename} The filename of the input program.

The fields are separated by one or more spaces, and each run of \textit{WSAT(CC)} is represented by a different line, delimited by the linefeed ("\n") character.

10.2 Output of Answer Sets

Like \texttt{aspps}, \textit{WSAT(CC)} supports the \texttt{-A} and \texttt{-S} command line options to control its output (see Section 9). However, rather than using the \texttt{-C} option to specify the number of solutions to display, \textit{WSAT(CC)} uses the \texttt{-t} option, which determines the number of attempts it will make at finding a solution. Since \textit{WSAT(CC)} finds solutions independently, some may be identical.

11 Examples

11.1 Graph Coloring

Data:

% This is a data file for the graph 3-coloring problem.
% The next line defines the four vertices of the input graph.

    vtx(1). vtx(2). vtx(3). vtx(4).

% The next line specifies the edges of the input graph.

    edge(1,4). edge(1,2). edge(3,2).

% Finally, the next line defines three available colors: r, g,
% and b.
    clr(r). clr(g). clr(b).

Program:
% We define the predicate color, to associate each vertex with a
% color.
    pred color(vtx, clr).
% We declare two variables of type vtx.
    var vtx X, Y.
% We declare one variable of type clr.
    var clr K.
% Each vertex has exactly one color.
    1{color(X, _)}1.
% Adjacent vertices cannot share the same color.
    color(X, K), color(Y, K), edge(X, Y) → .

11.2 N-Queens

Data:
% We declare a range of numbers, from 1 to the constant n.
    number[1..n].

Program:
% We declare a predicate q, which represents the placement of a
% queen on the chess board.
    pred q(number, number).
% We declare three variables of type number.
    var number R, C, I.
% There is exactly one queen per row.
1\{q(R,\_)}1.
% There is exactly one queen per column.

1\{q(\_,C)}1.
% There is at most one queen per diagonal.

% For the following comments, we assume the rows and columns of
% the chessboard are labeled from 1 to n starting from the
% bottom left corner.

% Allow at most one queen per diagonal in the upper
% left corner of the board.

{q(R+I-1,I): number(I)}1.
% Allow at most one queen per diagonal in the bottom
% right corner of the board.

{q(I,C+I-1): number(I)}1.
% Allow at most one queen per diagonal in the bottom
% left corner of the board.

{q(R-I+1,I): number(I)}1.
% Allow at most one queen per diagonal in the upper
% right corner of the board.

{q(n-I+1,C+I-1): number(I)}1.
A  **Aspps System Overview**

A.1  **Processing Programs in PS⁺ Format**

Problems expressed in PS⁺ format may, after grounding to PLᶜᶜ format with asppsgrnd, be solved using aspps, WSAT(CC), or another solver for which a conversion utility exists. These utilities are represented by Aspps2Y in the diagram below. The aspps system comes with programs for conversion from PLᶜᶜ format to PBS (aspps2pbs), WSAT(OIP) (aspps2oip), CPLEX (aspps2cplex), and CNF (aspps2cnf-bc and aspps2cnf-uc) formats.

The shaded portions of the following diagram represent modules which are not part of the aspps system.
A.2 Processing Ground Programs in Formats Other than PL$^{cc}$

Ground programs in formats other than PL$^{cc}$ may be converted to PL$^{cc}$ format for solving with aspps or WSAT(CC). The conversion utilities are represented by Y2aspps in the diagram below. The aspps system comes with pbs2aspps, a utility to convert PBS format to PL$^{cc}$ format.

The shaded portions of the following diagram represent modules which are not part of the aspps system.

![Diagram](image)

References


