Aspps 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

October 30, 2006

Contents

1 Introduction **1**

2 Syntax **2**
2.1 Basics . . . 2
2.1.1 Variables and Constants . . . 2
2.1.2 PS^+ atoms . . . 2
2.1.3 Clauses . . . 4
2.1.4 Arithmetic . . . 4
2.2 Data File . . . 4
2.2.1 Data Predicates — the instance-representation schema . . . 4
2.3 Program File . . . 5
2.3.1 Program predicates . . . 5
2.3.2 Variables . . . 6
2.3.3 Clauses . . . 6
2.4 Theories and Models . . . 6

3 Semantics **6**
3.1 Grounding . . . 6
3.1.1 Simplification . . . 7
3.2 Grounding Examples . . . 8
3.2.1 Atoms . . . 8
3.2.2 E-Atoms . . . 8
3.2.3 Conditional E-Atoms . . . 8
3.2.4 C-Atoms . . . 9
3.2.5 Clauses . . . 9
3.3 Models and Satisfaction . . . 10

4 Installation **10**
4.1 Obtaining the PS^+ software . . . 10
4.2 System Requirements . . . 10
4.3 Installing *psgrnd* on Unix Systems . . . 10
4.4 Installing *aspps* on Unix Systems . . . 11

5 Invoking *psgrnd* **11**

6 *psgrnd* Output Format **12**
6.1 Atoms . . . 13
6.2 C-atoms . . . 13
6.3 Clauses . . . 13

7 Invoking *aspps* **13**

8 *aspps* Output **14**
8.1 aspps.stat . . . 14
8.2 Output of Answer Sets . . . 15

9 Examples **15**
9.1 Graph Coloring . . . 15
9.2 N-Queens . . . 17
9.3 Wire Routing . . . 19
9.4 Missionaries and Cannibals . . . 24
9.5 Blocks-world Planning . . . 27
9.6 Vertex-cover Problem . . . 33

1 Introduction

The PS^+ software is an implementation of an answer-set programming formalism based on the logic of propositional schemata, PS^+ [1, 2]. The language of the logic PS^+ includes *constant*, *variable* and *predicate* symbols.

PS^+ *formulas* are clause-like expressions built of PS^+ *atoms*: first-order atoms, *existential* atoms and *weight* atoms. We typically refer to PS^+ formulas as PS^+ *clauses.* We always write PS^+ clauses 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 *instance* of a problem (input data) and P is a set of PS^+ clauses representing a *program* (an abstraction of the problem). The meaning of a PS^+ theory $T = (D, P)$ is given by a *family* of PS^+ *models* of T. PS^+ models of T are defined as models of a propositional theory in the language of propositional logic extended by weight atoms, denoted PL^{wa}. This theory is obtained by a form of grounding (we refer for details to [1, 2], Section 3.1).

The PS^+ software is designed to allow the programmer to:

1. model search problems as PS^+ theories, and
2. solve search problems modeled as PS^+ theories.

The tasks are supported by two programs: the program *psgrnd* that grounds ("compiles") PS^+ theories to a theory in the propositional logic PL^{wa}, and the program *aspps* that computes models of theories in the logic PL^{wa}. To model a search problem Π the programmer needs to design:

1. a relational schema to represent problem instances, we refer to it as an *instance-representation schema* (a relational schema specifies names of predicate symbols and their arities), and
2. a *program* P, a collection of PS^+ clauses capturing specifications of the problem Π.

To describe P, the programmer will use variable symbols and predicate symbols. The predicate symbols are of three types: (i) the predicate symbols from the instance-representation schema, (ii) predefined relation symbols, which we discuss below, and (iii) additional predicate symbols, called *program* predicate symbols, which the programmer needs to introduce.

The program needs to be designed so that for every instance of the problem Π, represented by a set D of ground atoms in the instance-representation schema, models of the PS^+ theory (D, P) encode solutions to the problem Π for the instance D. We refer to Section 9 for examples of search problems and their representations as programs in the logic PS^+.

Thus, to solve a problem Π, represented as a PS^+ program P, for an instance of Π, represented as a set D of ground atoms in the instance-representation schema, the programmer needs to perform two steps:

1. ground the PS^+ theory (D, P) by invoking the program *psgrnd*
2. compute models of the theory produced by *psgrnd* by invoking a PL^{wa} solver *aspps*.

In the remainder of the document we provide a detailed description of the language supported by the programs *psgrnd* and *aspps*, and illustrate its usage with examples.

2 Syntax

A precise description of the PS^+ syntax requiers a formal grammar. This section seeks to explain the PS^+ syntax in a more accessible, step-by-step manner.

2.1 Basics

2.1.1 Variables and Constants

PS^+ programs and data sets contain occurrences of *variable* and *constant* symbols. We always adhere to the convention that variable names start with upper-case letters and alpha-numeric constant names start with lower-case letters. We note that we also use integers as constants.

PS^+ software allows the programmer to use symbolic representations for some numeric (specifically, integer) constants and define their values at the command line (we discuss the details later). We refer to such constants as *symbolic* constants. As with constants, we use strings starting with lower-case letters to denote them.

The language does not contain function symbols. Thus, the only *terms* of the language are variables and constants.

2.1.2 PS^+ atoms

PS^+ programs and data sets contain occurrences of predicate symbols. These predicate symbols (i) come from the instance-representation schema, (ii) are introduced by the programmer, or (iii) are *predefined* predicate symbols to represent equality (`==`), inequality (`!=`), and arithmetic relations (`<=`, `>=`, `<`, and `>`). Predicate symbols allow the programmer to form *atoms*.

Simple atoms. They are expressions of the form:

```
atom-id(atom-arguments)
```

where `atom-id` is a string of one or more alphanumeric characters (A-Z, a-z, 0-9, _) beginning with a letter, and `atom-arguments` is a comma-delimited list of one or more constants or variables. An example of a simple atom is:

```
color(X,Y)
```

In the case of predefined predicate symbols, `==`, `!=`, `<=`, `>=`, `<`, or `>`, we write atoms using the standard infix notation. For example, we write:

```
Y!=Z
```

rather than `!=(Y,Z)`.

Existential atoms (or e-atoms). They are like simple atoms with one exception: their *atom-arguments* list contains an underscore character ('_') in place of one or more arguments. For example, the following expression is an e-atom:

```
color(X,_)
```

Conditional E-Atoms. These atoms are expressions of the form:

atom-id(*atom-arguments*)[*variable-list*]: *condition-list*

where *atom-id* and *atom-arguments* have the same meaning as before. The expression *variable-list* is a comma-delimited list of variables to be quantified existentially[1], and *condition-list* is a colon-delimited list of conditions to impose. These conditions may only involve predicate symbols that come from the instance-representation schema or are predefined. The precise meaning of conditional e-atoms is given later in the document. The following two examples illustrate the syntax of conditional e-atoms:

```
color(X,Y)[Y]: X<=5
color(X,Y): vtx(X): clr(Y)
```

Weight atoms or w-atoms. In this document we only focus on a special class of weight atoms — *cardinality* atoms (or simply, c-atoms). They are expressions of the form:

k{*atom-list*}*l*

where *atom-list* is a comma-delimited list of atoms, e-atoms, or conditional e-atoms, and `k` and `l` are optional constants or variables representing inclusive lower and upper bounds, respectively, on how many atoms in the list may be true (a precise meaning of these atoms is described later). Examples of a valid c-atoms are:

```
1{path(X,Y)[X]:edge(X,Y)}1
1{color(X,_)}2
1{path(X,Y)[X]:edge(X,Y),p(X):cond(X)}1
k{path(X,Y)[X]:edge(X,Y),p(X):cond(X)}l
```

In this last example, *k* and *l* stand for symbolic constants.

[1]See section 3.1 for an explanation of existential and universal quantification.

2.1.3 Clauses

Clauses in PS^+ take the form:

```
body -> head.
```

where ***body*** is a conjunction of atoms (of any type except e-atoms and conditional e-atoms) separated by commas (‘,’) and ***head*** is a disjunction of atoms (of any type) separated by pipes (‘|’). The head and body are separated by a dash followed by a greater than sign (“->”), and the clause is terminated with a period (‘.’). Thus, a standard clause is of the form

$atom_1, \ldots, atom_i$ -> $atom_{i+1} | \ldots | atom_j$.

Either of the body and head may be omitted. Some examples of valid clauses are

```
path(X,Y),path(Y,Z) -> path(X,Z) | 1{color(X,C): clr(C)}.
1{color(X,_)}1.
color(X,K),color(Y,K),edge(X,Y) -> .
```

2.1.4 Arithmetic

The *aspps* implementation predefines arithmetic operators such as +, -, *, and /. The functions abs, mod, max, and min are also available. We assign to these symbols their standard interpretation. The programmer can use these operators and functions to build arithmetic expressions and use them as arguments to predicate symbols. Examples of valid uses are:

```
X!=Y+Z
abs(X)>4
mod(X,2)==0
min(X,Y)<2
color(X+Y,C)
```

2.2 Data File

2.2.1 Data Predicates — the instance-representation schema

PS^+ data file provides an implicit definition of the instance-representation schema and an explicit representation of a particular problem instance for which the problem is to be solved. These definitions are of the following form:

```
data-id(data-arguments).
```

where ***data-id*** consists of one or more alphanumeric characters (A-Z, a-z, 0-9, _) beginning with a letter, and ***data-arguments*** consists of either a semicolon-delimited list of constants, or a *data range*. Data ranges are specified as a pair of integers or symbolic constants, separated with the string “..”.

Data predicate declarations are terminated with a single period, ‘.’. Examples of valid data predicate declarations are:

```
clr(red).
clr(green).
clr(blue).
```

or, the equivalent:

```
clr(red;green;blue).
```

Versions 2003.06.04 and later of *psgrnd* support the following range specification for predicate declaration[2]:

```
num(1..100).
```

This is meant to represent one hundred atomic expressions:

```
num(1).
⋮
num(100).
```

2.3 Program File

The program file consists of a preamble containing declarations of program predicate and of variables, followed by zero or more PS^+ clauses, defining the problem.

2.3.1 Program predicates

Program predicate declarations are of the form:

pred *predicate-name*(*data-id*$_1$,...,*data-id*$_n$).

where *predicate-name* is a user-specified string naming the predicate, and each *data-id*$_i$ is a name of a unary predicate specified in the data file. Examples of valid program predicate declarations are:

```
pred color(vtx,clr).
pred visited(vtx).
```

They assume that predicates *color* and *vtx* are part of the instance-representation schema and are defined in the data file.

[2]Older versions of *psgrnd* require the use of square brackets ('[' and ']') when specifying ranges.

2.3.2 Variables

Variables used in the program file are declared as follows:

```
var data-id variable-name_1,...,variable-name_n.
```

where *data-id* is a unary predicate specified in the data file, and each *variable-name$_i$* is a user-specified string of one or more alphanumeric characters which is used as a variable name. As we noted these names will start with upper-case letters. Two examples of variable declarations are:

```
var num X,Y.
var vtx V.
```

2.3.3 Clauses

Clauses follow the predicate and variable declarations and are constructed as described in section 2.1.3.

2.4 Theories and Models

A theory in the logic PS^+ is a pair (D, P), where D is a data file and P is a program file. We often refer to PS^+ theories as *data-program pairs*.

3 Semantics

The meaning of a PS^+ theory (D, P) is given by a *family* of PS^+ *models* of (D, P). A set of ground atoms, selected from the set of all ground atoms built of program predicates defined in P and of constants listed in D, is a PS^+ model of (D, P) if it is a model of the propositional theory obtained by grounding (D, P) and then simplifying the result [1, 2]. We denote this theory by $gr^s(D, P)$. The next two subsections describe grounding and simplification.

3.1 Grounding

All clauses in PS^+ programs are grounded separately using the following steps.

1. First, each underscore is replaced by a variable of the proper type. For example, the e-atom `p(_)` will become `p(X)[X]`, where `X` is a unique variable name.

2. Variables listed in square brackets alongside conditional e-atoms (including those which have replaced underscores) are existential. All the remaining variables are universal.

3. Next, we instantiate clauses by taking all possible substitutions for universal variables (as defined by their domain, in the data file) and creating new instances of the clause, one per substitution.

4. Once we have all the instantiations, only existential variables remain.

5. For each resulting clause, e-atoms are expanded to a disjunction of atoms containing all possible substitutions for each existential variable, as limited by each variable's respective domain.

6. Conditional e-atoms are likewise expanded to such a disjunction, but the values of their existential variables are constrained to those that satisfy their conditions.

7. E-atoms and conditional e-atoms inside of c-atoms are expanded to a set rather than a disjunction.

8. Next, arithmetic expressions such as `+`, `-`, `*`, `/`, `abs()`, and `mod()` are evaluated.

9. Built-in predicates and data predicates are then evaluated, as are comparisons using relational operators such as $<$, $>$, $<=$, $>=$, and $==$. (All but $==$ apply only to integers; $==$ may apply to arbitrary constants as well.)

3.1.1 Simplification

Finally, the process of simplification begins. Program predicates with invalid parameters (any values outside the domain of the data type) are marked $false$. Program predicates with valid parameters are marked $true$.

Built-in atoms and data atoms, once evaluated, enable certain simplifications.

- $True$ atoms which occur in the body of a clause are eliminated.
- $False$ atoms in the head of a clause are likewise eliminated.
- $False$ atoms in the body of a clause cause the entire clause to be eliminated.
- $True$ atoms in the head of a clause also cause the entire clause to be eliminated.
- When an atom within a c-atom is determined to be $true$, the atom is eliminated and the c-atom bounds are decremented by one.
- If an atom within a c-atom is $false$, it is likewise eliminated, but the bounds are preserved.

The resulting clauses are output as a ground propositional theory $gr^s(D, P)$ in the language of the logic PL^{wa}. Formulas of this logic look like propositional clauses written in the *implication* notation. That is, they are implications with the antecedent being a conjunction of atoms and propositional c-atoms, and the consequent being a disjunction of atoms and propositional c-atoms. In Section 3.3 we specify when such clauses are satisfied by a set of ground atoms. But first we illustrate grounding and simplification with examples.

3.2 Grounding Examples

3.2.1 Atoms

Given the data file:

```
num(1..3).
```

and the rule file:

```
pred atom(num).

atom(1).
atom(3).
```

The following grounding is obtained:

```
atom(1).
atom(3).
```

3.2.2 E-Atoms

Given the data file:

```
num(1..2).
clr(r).
clr(g).
clr(b).
```

and the rule file:

```
pred color(num,clr).
var num X.

color(X,_).
```

The following grounding is obtained:

```
color(1,r)|color(1,g)|color(1,b).
color(2,r)|color(2,g)|color(2,b).
```

3.2.3 Conditional E-Atoms

Given the data file:

```
num(1..3).
```

and the rule file:

```
pred lessthan(num,num).
var num X,Y.

lessthan(X,Y)[Y]: X<Y.
```

The following grounding is obtained:

```
lessthan(1,2)|lessthan(1,3).
lessthan(2,3).
```

3.2.4 C-Atoms

Given the data file:

```
num(1..2).
clr(r).
clr(g).
clr(b).
```

and the rule file:

```
pred color(num,clr).
var num X.

1{color(X,_)}2.
```

The following grounding is obtained:

```
1{color(1,r),color(1,g),color(1,b)}2.
1{color(2,r),color(2,g),color(2,b)}2.
```

3.2.5 Clauses

Given the data file:

```
num(1..3).
clr(r).
clr(g).
clr(b).
```

and the rule file:

```
pred color(num,clr).
var num X.
var clr K.

1{color(X,K)[K]}1.
color(X,K),X<=2 -> color(X,r).
color(X,K),X>2 -> color(X,b).
```

The following grounding is obtained:

```
1{color(1,r),color(1,g),color(1,b)}1.
1{color(2,r),color(2,g),color(2,b)}1.
1{color(3,r),color(3,g),color(3,b)}1.
```

```
color(1,r) -> color(1,r).
color(1,g) -> color(1,r).
color(1,b) -> color(1,r).
color(2,r) -> color(2,r).
color(2,g) -> color(2,r).
color(2,b) -> color(2,r).
color(3,r) -> color(3,b).
color(3,g) -> color(3,b).
color(3,b) -> color(3,b).
```

3.3 Models and Satisfaction

A model of a grounded and simplified PS^+ theory $T = (D, P)$ (and so, of the theory itself) is an assignment of truth values to atoms such that all clauses in $gr^s(D, P)$ are satisfied.

- A clause is satisfied in all cases *except* when its body is $true$ (possibly because it is empty) and its head is $false$ (possibly because it is empty).
- The body of a clause, as a conjunction, is $true$ only when all of its atoms and c-atoms are $true$.
- The head of a clause, a disjunction, is $true$ when at least one of its atoms or c-atoms is $true$.
- A c-atom is satisfied when the number of $true$ atoms it contains is within its lower and upper bounds.

4 Installation

4.1 Obtaining the PS^+ software

The programs *psgrnd* and *aspps* and several related programs and utilities can be obtained at `http://www.cs.uky.edu/ai/aspps/`.

4.2 System Requirements

The programs *psgrnd* and *aspps* 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.

4.3 Installing *psgrnd* on Unix Systems

Installing *psgrnd* is straightforward:

1. Type `gunzip -c psgrnd.`*XXXX.XX.XX*`.tar.gz | tar xvf -` to extract the source code archive (where *XXXX.XX.XX* is the version-specific date of *psgrnd*).
2. Type `cd Psgrnd`
3. Type `make` to compile *psgrnd*.
4. Install the compiled program by placing it in a directory for executable files. For example, type `cp psgrnd /usr/local/bin/`

4.4 Installing *aspps* on Unix Systems

The installation process for *aspps* is similar.

1. Type `gunzip -c aspps.`*XXXX.XX.XX*`.tar.gz | tar xvf -` to extract the source code archive (where *XXXX.XX.XX* is the version-specific date of *aspps*).
2. Type `cd aspps`
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/`

5 Invoking *psgrnd*

The grounding of PS^+ theories is performed by the program *psgrnd*. The required input to execute *psgrnd* is a single program file, one or more data files, and values for symbolic constants to be 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 `-o` command line option is used). The format of the *psgrnd* output is covered in Section

psgrnd has the following command line options:

> `psgrnd -r` *rfile* `-d` $dfile_1$ `[`$dfile_2$ `...` $dfile_m$`] [-c` c_1`=`v_1 `[`c_2`=`v_2 `...` c_n`=`v_n`]] [-o [`*theoryfile*`]] [-C]`

Required Arguments:

`-r` *rfile*

> *rfile* is the file describing the problem (the program file). There must be exactly one program file.

`-d` $dfile_1$ `[`$dfile_2$ `...` $dfile_m$`]`

> Each $dfile_i$ is a data file containing data that will be used to instantiate the theory.

Optional Arguments:

`-c` c_1=v_1 [c_2=v_2 ... c_n=v_n]

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

`-o [`*`theoryfile`*`]`

> If *`theoryfile`* is specified, *psgrnd* 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 *psgrnd*'s output to DIMACS CNF format [3]. It can only be used if no weight (in particular, 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 *psgrnd*:

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

6 *psgrnd* Output Format

The ground programs generated by *psgrnd* 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 <num of atoms> <num of clauses>
  ```

- Clauses are terminated by the end of the line ('`\n`'), not with a '`0`'.
- C-atoms are preserved and represented using special notation.

6.1 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.

6.2 C-atoms

A c-atom $\texttt{k\{ a}_1 \texttt{ , ... , a}_n \texttt{ \}m}$ is represented after grounding as:

$$\texttt{\{ k m int}_{a_1} \texttt{ ... int}_{a_n} \texttt{ \}}$$

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

6.3 Clauses

A clause $\texttt{A}_1 \texttt{ , ... , A}_m \texttt{ -> B}_1 \texttt{ | ... | B}_n$ is expressed after grounding as:

$$r(A_1) \texttt{ ... } r(A_m) \texttt{ , } r(B_1) \texttt{ ... } r(B_n)$$

where $r(A) = \texttt{int}_a$, if $A = a$, for some ground atom a, and $r(A) = \texttt{\{ k m int}_{a_1} \texttt{ ... int}_{a_n} \texttt{ \}}$, if $A = \texttt{k\{ a}_1 \texttt{ , ... , a}_n \texttt{ \}m}$ is a cardinality atom (as before, we write $\texttt{int}_a$ 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.

7 Invoking *aspps*

The solver, *aspps*, is used to compute models of the ground PL^{wa} theory produced by *psgrnd*. 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`.

The program *aspps* has the following command line options:

aspps -f *filename* [-A] [-P] [-C [x]] [-L [x]] [-S *name*$_1$ [*name*$_2$ ... *name*$_m$]] [-r]

Required Arguments:
-f *filename*

Where *filename* is the name of the file containing a ground theory in PL^{wa} format.

Optional Arguments:

`-A`

Prints atoms that are true in the computed model, in readable form.

`-C [x]`

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 x is not specified, *aspps* finds all solutions. Finally, if the flag is used and x is specified (where x is a positive integer), *aspps* stops after finding x solutions or exhausting the whole search space, whichever comes first.

`-L [x]`

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

`-S` $name_1$ [$name_2$ ... $name_m$]

Shows positive atoms built of predicates $name_i$.

`-r`

For versions of aspps that allow restarts (for instance, version of 05/17/2005), turns restarts of

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:

`Var`	The number of variables in the input program.
`Clause`	The number of clauses in the input program.
`Sat`	The status of the theory: `sat` or `unsat`, respectively
`CPU Time`	The amount of CPU time used, in seconds.
`Branch`	The number of branch points.
`Bcktrck`	The number of times *aspps* backtracked.
`Count`	The number of solutions found.
`Filename`	The filename of the input program.

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 9 will cause *aspps* to generate lines similar to the following:

```
Var  Clause  Sat  CPU Time  Branch  Bcktrck  Count  Filename
16   13      sat  0         22      24        24     gc.aspps
```

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^{wa} program), *aspps* will output the following:

```
aspps.2003.01.09

Atoms determined during grounding:

Answer-set 1:
 color(1,r)
 color(2,g)
 color(3,r)
 color(4,g)
```

9 Examples

This section contains examples of common problems and puzzles implemented in PS^+.

9.1 Graph Coloring

Given an undirected graph, the goal is to assign colors to its vertices such that no two vertices connected by an edge share the same color.

Data File

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;
% its parameters must be vertices as defined above.

    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 File

The single program predicate $color(X, K)$ represents the fact that vertex X is assigned color K.

```
% 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) -> .
```

***aspps* and *psgrnd* Output**

```
$ psgrnd -r 1_gc.rule -d 1_gc.data -o 1_gc.aspps
Version: psgrnd 7-Jul-2005
SUCCESS: Created file "1_gc.aspps"
```

```
$ aspps -f 1_gc.aspps -A

aspps.2003.06.04

Atoms determined during grounding:

Answer-set 1:
 color(1,b)
 color(2,g)
 color(3,b)
 color(4,g)
```

Aspps.stat Information

```
Variable  Clauses   Sat   CPU Time  Branch    Bcktrck  Count     Filename
16        13        sat   0         4         0         1         1_gc.aspps
```

9.2 N-Queens

Given an $n \times n$ chessboard, the goal is to place on it n queens so that no two queens attack each other (that is, no two queens are in the same row, column, or in the same diagonal).

Data File

Data predicate *number* specifies the number of queens and the dimensions of the board.

```
% We declare a range of numbers, from 1 to the constant n.
% Constant n will be specified on the command line.

    number(1..n).
```

Program File

A simple encoding of the N-Queens problem is used here for the sake of clarity.

The main program predicate is *queen*. Atom $queen(R, C)$ represents the fact that a queen is placed in the position (R, C).

```
% We declare a predicate queen, which represents the placement of a
% queen on the chess board.

    pred queen(number,number).

% We declare three variables of type number.

    var number R,C,I.
```

```
% There is exactly one queen per row.

    1{queen(R,_)}1.

% There is exactly one queen per column.

    1{queen(_,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.

    {queen(R+I-1,I)[I]}1.

% Allow at most one queen per diagonal in the bottom
% right corner of the board.

    {queen(I,C+I-1)[I]}1.

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

    {queen(R-I+1,I)[I]}1.

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

    {queen(n-I+1,C+I-1)[I]}1.
```

Aspps **and** *Asppsgrnd* **Output**

```
$ psgrnd -r 2_nq.rule -d 2_nq.data -c n=8 -o 2_nq.aspps
Version: psgrnd 7-Jul-2005
SUCCESS: Created file "2_nq.aspps"
$ aspps -f 2_nq.aspps -A

aspps.2003.06.04

Atoms determined during grounding:

Answer-set 1:
 queen(1,7)
```

```
queen(2,1)
queen(3,3)
queen(4,8)
queen(5,6)
queen(6,4)
queen(7,2)
queen(8,5)
```

Aspps.stat Information

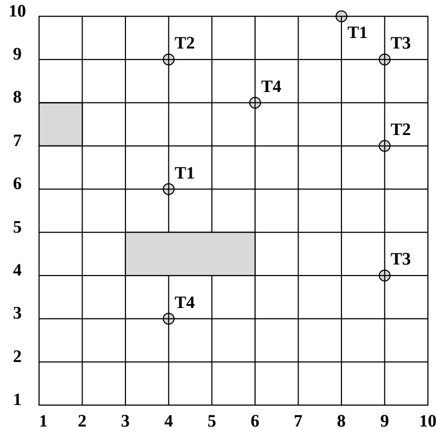
```
Variable   Clauses    Sat    CPU Time   Branch     Bcktrck   Count      Filename
112        48         sat    0          4          0          1          2_nq.aspps
```

9.3 Wire Routing

Given pairs of terminals on a grid and a set of obstacles, the goal is to connect terminals by wires so that the wires:

- do not intersect
- do not enter obstacles
- do not share grid edges

The provided data file for this example models the following wire routing configuration:



Data File

```
% We define four wires.

    wire(1). wire(2). wire(3). wire(4).
```

```
% We define the number of columns and rows

    crd(1). crd(2). crd(3). crd(4). crd(5).
    crd(6). crd(7). crd(8). crd(9). crd(10).

% We define terminal locations.
% terminal(X,Y,Z) means the end of wire Z is in (X,Y).

    terminal(6,4,1). terminal(10,8,1).
    terminal(7,9,2). terminal(9,4,2).
    terminal(4,9,3). terminal(9,9,3).
    terminal(8,6,4). terminal(3,4,4).

% We define blocked points

    blocked(4,3). blocked(4,4). blocked(4,5). blocked(4,6).
    blocked(5,3). blocked(5,4). blocked(5,5). blocked(5,6).
    blocked(7,1). blocked(7,2). blocked(8,1). blocked(8,2).
```

Program File

```
% Declare program predicate path: point (I,J) is on wire W

    pred path(crd, crd, wire).

% Declare variables

    var crd I, J, K, L, M, N.
    var wire W.

% Terminal points are on the path

    terminal(I,J,W) -> path(I,J,W).

% Terminal points have exactly one adjacent point on the path

    terminal(I,J,W) -> 1 { path(L,M,W)[L,M]: abs(I-L)+abs(J-M)==1 } 1.

% All other points on the path have exactly two adjacent points

    path(I,J,W) -> terminal(I,J,W) |
                   2 { path(L,M,W)[L,M]: abs(I-L)+abs(J-M)==1 } 2.

% Only one wire can pass through any point
```

```
    { path(I,J,W)[W] } 1.

% Blocked points are unavailable

    blocked(I,J), path(I,J,W) -> .
```

Aspps **and** *Asppsgrnd* **Output**

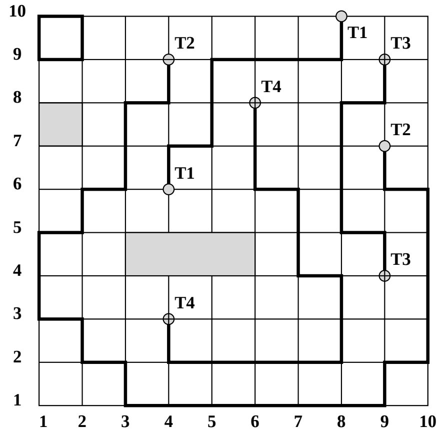
```
$ psgrnd -r 3_wr.rule -d 3_wr.data -o 3_wr.aspps
Version: psgrnd 7-Jul-2005
WARNING: The following program variables were declared but never used:  K, N

SUCCESS: Created file "3_wr.aspps"
$ aspps -f 3_wr.aspps -A

aspps.2003.06.04

Atoms determined during grounding:

Answer-set 1:
 path(1,3,2)
 path(1,4,2)
 path(1,5,2)
 path(1,7,2)
 path(1,8,2)
 path(1,9,2)
 path(2,2,2)
 path(2,3,2)
 path(2,5,2)
 path(2,6,2)
 path(2,7,2)
 path(2,9,2)
 path(2,10,2)
 path(3,1,2)
 path(3,2,2)
 path(3,4,4)
 path(3,5,4)
 path(3,6,4)
 path(3,7,4)
 path(3,10,2)
 path(4,1,2)
 path(4,7,4)
 path(4,8,3)
 path(4,9,3)
 path(4,10,2)
 path(5,1,2)
 path(5,2,2)
```

```
path(5,7,4)
path(5,8,3)
path(5,9,2)
path(5,10,2)
path(6,2,2)
path(6,3,2)
path(6,4,1)
path(6,6,4)
path(6,7,4)
path(6,8,3)
path(6,9,2)
path(7,3,2)
path(7,4,1)
path(7,6,4)
path(7,8,3)
path(7,9,2)
path(8,3,2)
path(8,4,1)
path(8,5,1)
path(8,6,4)
path(8,8,3)
path(8,9,3)
path(9,1,1)
path(9,2,1)
path(9,3,2)
path(9,4,2)
path(9,5,1)
path(9,6,1)
path(9,7,1)
path(9,8,1)
path(9,9,3)
path(10,1,1)
path(10,2,1)
path(10,8,1)
```

Aspps.stat Information

Variable	Clauses	Sat	CPU Time	Branch	Bcktrck	Count	Filename
900	556	sat	24.04	318801	325275	1	3_wr.aspps

The program above provides the following solution:



There are several problems with this solution. Meandering paths between terminals result in unnecessarily long wires. Loops also may occur, as seen in the upper left corner of the grid. These problems can be eliminated with the addition of further constraints to the wire-routing program, however.

Program File (continued)

```
% At most m points on each wire

    { path(I,J,W)[I,J] } m.

% Do not choose points at distance m or more from a terminal point

    terminal(I,J,W), path(L,M,W), m < abs(I-L)+abs(J-M) -> .

% Eliminate cycles of length 4

    path(I,J,W), path(I+1,J,W), path(I+1,J+1,W), path(I,J+1,W) -> .

% Limit search space to rectangle formed by terminal points
% (or its relaxation).  The rectangle is expanded by k units.

    terminal(I,J,W), terminal(M,N,W), path(K,L,W) -> K >= min(I,M)-k.
    terminal(I,J,W), terminal(M,N,W), path(K,L,W) -> K <= max(I,M)+k.
    terminal(I,J,W), terminal(M,N,W), path(K,L,W) -> L >= min(J,N)-k.
    terminal(I,J,W), terminal(M,N,W), path(K,L,W) -> L <= max(J,N)+k.
```

Aspps **and** *Asppsgrnd* **Output**
This output is incorrect. To be fixed.

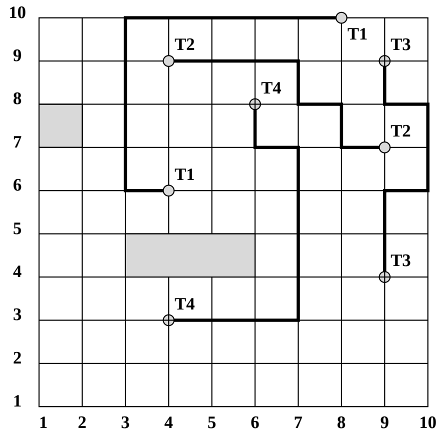
```
$ psgrnd -r 3_wr.rule -d 3_wr.data -c m=11 k=1 -o 3_wr.aspps
Version: psgrnd 7-Jul-2005
```

```
SUCCESS: Created file "3_wr.aspps"
$ aspps -f 3_wr.aspps -A
aspps.2003.06.04
```

Aspps.stat Information

Variable	Clauses	Sat	CPU Time	Branch	Bcktrck	Count	Filename
904	2666	unsat	0	0	0	0	3_wr.aspps

The revised program with these extra clauses provides the following solution:



As one can see, the problems are no longer present.

9.4 Missionaries and Cannibals

There are three cannibals and three missionaries on the left bank of a river. They have a boat. The boat has capacity 2. If at any time there are more cannibals than missionaries on either bank, the cannibals eat the missionaries. The goal is to arrange a way of moving all 6 of them to the right bank. The problem of missionaries and cannibals is a classic AI puzzle that received much attention [Lif00].

Data File

This example instance requires 11 moves to solve.

```
% Specify time range (number of moves) in which the solution
% is to be found

    time(0..t).

% Specify allowed number of cannibals (or missionaries) on a bank

    num(0..3).
```

```
% 0 - boat on the left bank
% 1 - boat on the right bank

    boat(0..1).

% Initial state

    instate(3,3,0).
```

Program File

The two key predicates are *state* and *move*. Atom $state(T, CC, MM, B)$ has the following meaning: at time T there are CC cannibals and MM missionaries on the left bank. $B = 0$ means that the boat is on the left bank, $B = 1$ means that the boat is on the right bank. Atom $move(T, C, M)$ means that at time T, C cannibals and M missionaries are moving to the opposite bank of the river (the direction depends on the location of the boat).

```
% Declare predicates

    pred state(time,num,num,boat).
    pred move(time,num,num).

% Declare variables

    var time T.
    var num M,C,MM,CC.
    var boat B.

% At time 0, the state coincides with the initial state given
% by the input

    state(0,CC,MM,B) -> instate(CC,MM,B).

% Select a configuration at time T

    1{state(T,CC,MM,B)[CC,MM,B]}1.

% Select a move for time T

    T<t -> 1{move(T,C,M)[C,M]}1.

% No moves at time t (the plan ends at time t)

    move(t,C,M) -> .

% Enforce the boat capacity costraint
```

```
    move(T,C,M) -> M+C<=2.

% At least one person must be moving to the other bank

    move(T,C,M) -> 0<C+M.

% If the boat is on the left bank no more than CC cannibals
% and no more than MM missionaries may be chosen to move to
% the other side

    state(T,CC,MM,0),move(T,C,M) -> C<=CC.
    state(T,CC,MM,0),move(T,C,M) -> M<=MM.

% If the boat is on the right bank no more than 3-CC cannibals
% (this is how many of them ae on the right bank) and no more
% than 3-MM missionaries may be chosen to move to the other side

    state(T,CC,MM,1),move(T,C,M) -> C<=3-CC.
    state(T,CC,MM,1),move(T,C,M) -> M<=3-MM.

% Ensure that a move is not immediately "undone"

    0<T, T<t, move(T,C,M), move(T-1,C,M) -> .

% Relate the next state to the previous one and to the
% move chosen

    T<t, move(T,C,M), state(T,CC,MM,0) -> state(T+1,CC-C,MM-M,1).
    T<t, move(T,C,M), state(T,CC,MM,1) -> state(T+1,CC+C,MM+M,0).

% Eliminate states not safe for the missionaries

    state(T,CC,MM,B), MM!=0, CC > MM -> .
    state(T,CC,MM,B), MM!=3, CC < MM -> .

% At time t the goal state must be reached

    state(t,0,0,1).
```

Aspps **and** *Asppsgrnd* **Output**

```
$ psgrnd -r 5_mc.rule -d 5_mc.data -c t=11 -o 5_mc.aspps
Version: psgrnd 7-Jul-2005
SUCCESS: Created file "5_mc.aspps"
$ aspps -f 5_mc.aspps -S move
```

```
aspps.2003.06.04

Atoms determined during grounding:
 move

Answer-set 1:
 move(0,1,1)
 move(1,0,1)
 move(2,2,0)
 move(3,1,0)
 move(4,0,2)
 move(5,1,1)
 move(6,0,2)
 move(7,1,0)
 move(8,2,0)
 move(9,1,0)
 move(10,2,0)
```

Aspps.stat Information

Variable	Clauses	Sat	CPU Time	Branch	Bcktrck	Count	Filename
599	10747	sat	0.02	6	5	1	5_mc.aspps

9.5 Blocks-world Planning

Blocks are arranged in stacks on the floor. Each block is located on exactly one block or on the floor. No two blocks are placed on the same block. A block that is on top of its stack can be moved and placed on the top of another stack or on the floor. Find a plan (a sequence of actions) to rearrange the blocks into a desired final configuration. Both sequential (only one block moved in a step) and concurrent (allows for several blocks to be moved in the same step) versions are of interest. This is a classic planning problem in AI.

Data File

Data predicate *time* specifies the time horizon (constant t is entered from the command line). Data predicate block defines the set of blocks (constant k is entered from the command line). Data predicate *on0* defines the initial configuration. Atom $on0(A, B)$ means that block A is (initially) on block B. Similarly, data predicate *onF* provides a specification for the goal configuration. Atom $onF(A, B)$ means that in the goal configuration block A must be on block B. The specification of the goal configuration may be incomplete. The data file given here is the example `large.c`, proposed by Kautz and Selman (`http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`).

```
% We define the time horizon
```

```
    time(0..t).

% We define the number of blocks

    block(0..15).

% Initial configuration

    on0(13,0).  on0(12,13).  on0(1,12).  on0(2,1).  on0(3,2).
    on0(15,0).  on0(14,15).  on0(4,14).  on0(5,4).  on0(10,5).
    on0(11,10).  on0(6,0).  on0(7,6).  on0(8,7).  on0(9,8).

% Partial description of the goal configuration

    onF(5,10).  onF(1,5).  onF(14,1).  onF(9,4).  onF(8,9).
    onF(13,8).  onF(15,13).  onF(11,7).  onF(3,11).  onF(2,3).
    onF(12,2).
```

Program File

Key program predicates are *on*, *move* and *top*. Atom $on(T, A, B)$ means that at time T block A is on block B. Atom $move(T, A)$ represents the fact that at time T, block A is one of the blocks to be moved. Finally, atom $top(T.A)$ means that at time T block A is at the top of its stack of blocks.

```
% Declare predicates

    pred on(time,block,block).
    pred move(time,block).
    pred top(time,block).

% Declare variables

    var time T.
    var block A,B,C.

% Specify an arrangement of blocks at time T

    % Each block other than the floor is on top of exactly one
    % block

        0<A -> 1 { on(T,A,B)[B] } 1.

    % Each block other than the floor is either a top block or
    % is under exactly one block or is the top block

        0<A -> 1{top(T,A); on(T,B,A)[B]}1.
```

```
    % floor is never on any other block

        on(T,0,A) -> .

    % floor is never a top

        top(T,0) -> .


% The state at time 0 must coincide with the initial configuration

    on(0,A,B) -> on0(A,B).
    on0(A,B) -> on(0,A,B).

% The state at time t must be consistent with the specification
% of the goal

    onF(A,B) -> on(t,A,B).

% Constraints on moves

    % Floor never moves

        move(T,0) -> .

    % Moved blocks are restricted to top blocks.

        on(T,B,A), move(T,A) -> .

    % No move is made at time t.

        move(t,A) -> .

% A block cannot be moved on top of a block that is being moved
% in the same move

    move(T,A),move(T,B),on(T+1,A,B) -> .

% No block is directly moved back where it was

    move(T,A), on(T,A,B), on(T+2,A,B) -> .

% No block moves on the same location it was in

    move(T,A), on(T,A,B), on(T+1,A,B) -> .
```

```
% Compute the new state

    % None of these constraints is necessary. But when put together
    % they seem to work best.

        T<t, on(T,A,B) -> 1{on(T+1,A,B); move(T,A)}1.
        on(T+1,A,B) -> 1{move(T,A); on(T,A,B)}1.
```

Aspps **and** *Asppsgrnd* **Output**

```
$ psgrnd -r 6_bwp.rule -d 6_bwp.data -c t=8 -o 6_bwp.aspps
Version: psgrnd 7-Jul-2005
SUCCESS: Created file "6_bwp.aspps"
$ aspps -f 6_bwp.aspps -S move

aspps.2003.06.04

Atoms determined during grounding:
on(0,1,12)
on(0,2,1)
on(0,3,2)
on(0,4,14)
on(0,5,4)
on(0,6,0)
on(0,7,6)
on(0,8,7)
on(0,9,8)
on(0,10,5)
on(0,11,10)
on(0,12,13)
on(0,13,0)
on(0,14,15)
on(0,15,0)
on(1,1,12)
on(1,2,1)
on(1,4,14)
on(1,5,4)
on(1,6,0)
on(1,7,6)
on(1,8,7)
on(1,10,5)
on(1,12,13)
on(1,13,0)
on(1,14,15)
on(1,15,0)
on(2,1,12)
```

```
on(2,4,14)
on(2,5,4)
on(2,6,0)
on(2,7,6)
on(2,12,13)
on(2,13,0)
on(2,14,15)
on(2,15,0)
on(3,4,14)
on(3,6,0)
on(3,12,13)
on(3,13,0)
on(3,14,15)
on(3,15,0)
on(4,13,0)
on(4,14,15)
on(4,15,0)
on(5,9,4)
on(5,11,7)
on(5,15,0)
on(6,3,11)
on(6,5,10)
on(6,8,9)
on(6,9,4)
on(6,11,7)
on(7,1,5)
on(7,2,3)
on(7,3,11)
on(7,5,10)
on(7,8,9)
on(7,9,4)
on(7,11,7)
on(7,13,8)
on(8,1,5)
on(8,2,3)
on(8,3,11)
on(8,5,10)
on(8,8,9)
on(8,9,4)
on(8,11,7)
on(8,12,2)
on(8,13,8)
on(8,14,1)
on(8,15,13)
move(0,11)
move(1,10)
```

```
move(2,5)
move(3,4)
move(4,9)
move(5,8)
move(6,13)
move(7,15)
top(0,3)
top(0,9)
top(0,11)
top(1,10)
top(1,11)
top(2,5)
top(2,10)
top(3,4)
top(3,5)
top(4,4)
top(4,9)
top(5,8)
top(5,9)
top(6,8)
top(6,13)
top(7,13)
top(7,15)
top(8,15)
 move

Answer-set 1:
 move(0,3)
 move(0,9)
 move(1,2)
 move(1,3)
 move(1,8)
 move(2,1)
 move(2,3)
 move(2,7)
 move(2,9)
 move(3,3)
 move(3,6)
 move(3,7)
 move(3,9)
 move(3,11)
 move(3,12)
 move(4,6)
 move(4,8)
 move(4,11)
 move(4,12)
```

```
move(4,13)
move(4,14)
move(5,3)
move(5,5)
move(5,6)
move(5,13)
move(5,15)
move(6,1)
move(6,2)
move(6,15)
move(7,12)
move(7,14)
```

Aspps.stat Information

Variable	Clauses	Sat	CPU Time	Branch	Bcktrck	Count	Filename
1620	2185	sat	0.34	922	920	1	6_bwp.aspps

9.6 Vertex-cover Problem

A set of vertices of an undirected graph is called a *vertex cover* if every edge in the graph has at least one of its ends in the set. The problem is, given an undirected graph G and an integer k, to compute a vertex cover of G with no more than k vertices (if such a cover exists). The vertex-cover problem is another classic NP-complete problem. It is also a difficult test problem for search algorithms.

The data file defines the vertex set of the graph. It also lists all the edges of the graph as facts $edge(a, b)$.

Data File

This example instance has a solution when k equals 2.

```
% Define the vertex set of the graph.

    vtx (1..5).

% Define the edges connecting the vertices.

    edge(1,3). edge(3,2). edge(3,4). edge(4,5).
```

Program File

The main (and only) program predicate is *incover*. Atom $incover(X)$ represents the fact that the vertex X is in a vertex cover.

```
% Declare program predicates

    pred incover(vtx).
```

```
% Declare variables

    var vtx X, Y.

% Select a set of vertices for a vertex cover

    { incover(X)[X] } k.

% Enforce the vertex-cover constraint:at least one end
% of each edge must be in the cover

    edge(X,Y) -> incover(X) | incover(Y).
```

Aspps **and** *Asppsgrnd* **Output**

```
$ psgrnd -r 8_vc.rule -d 8_vc.data -c k=2 -o 8_vc.aspps
Version: psgrnd 7-Jul-2005
SUCCESS: Created file "8_vc.aspps"
$ aspps -f 8_vc.aspps -A

aspps.2003.06.04

Atoms determined during grounding:

Answer-set 1:
 incover(3)
 incover(4)
```

Aspps.stat Information

Variable	Clauses	Sat	CPU Time	Branch	Bcktrck	Count	Filename
6	5	sat	0	2	0	1	8_vc.aspps

References

[1] D. East and M. Truszczyński. *aspps* – An Implementation of Answer-set Programming with Propositional Schemata. In *Proceedings of Logic Programming and Nonmonotonic Reasoning Conference, LPNMR 2001*, LNAI 2173, pages 402–405, Springer Verlag, 2001.

[2] D. East and M. Truszczyński. Propositional Satisfiability in Answer-set Programming. In *Proceedings of Joint German/Austrian Conference on Artificial Intelligence, KI'2001*, LNAI 2174, pages 138–153, Springer Verlag, 2001.

[3] `http://www.satlib.org/`

[Lif00] V. Lifschitz. Missionaries and cannibals in the causal calculator. In *Principles of Knowledge Representation and Reasoning, Proceedings of the Seventh International Conference (KR2000)*, pages 85 – 96. Morgan Kaufmann Publishers, 2000.