# AI2 Module 1 Practical 1

Bob Fisher
School of Informatics

Autumn 2004

**Abstract**

This document describes the first assignment (of 2) for assessment on AI2 Module 1. The main goal is to develop and compare two approaches to finding truth values that satisfy an expression.

## Task Background

Lecture 7 introduced the ideas behind constraint logic programming as implemented in Prolog. This assignment implements a constraint logic approach and an exhaustive search approach to finding the truth values that satisfy a logical expression. By this, we mean finding the values of `t` or `f` for each elementary proposition in an expression.

The key input data structure is a logical expression, which (here) is going to be a conjunction of terms $t_1 \wedge t_2 \wedge \ldots \wedge t_n$ (which means $t_1$ AND $t_2$ AND ... AND $t_n$), where each term $t_i$ is the disjunction of elementary terms. Each disjunction has the form $e_1 \vee e_2 \vee \ldots \vee e_n$ (which means $e_1$ OR $e_2$ OR ... OR $e_m$), where each $e_i$ is either an elementary proposition $v$ or the negation of an elementary proposition $\neg v$.

Thus, valid short expressions are $(a \vee b) \wedge (\neg a \vee b \vee c)$ and $(a \vee \neg b \vee c) \wedge (\neg c)$. A longer expression is: $(a \vee \neg b \vee c) \wedge (b \vee \neg c) \wedge (a \vee d \vee e) \wedge (\neg a \vee g \vee e) \wedge (\neg g \vee f \vee \neg e) \wedge (\neg b \vee \neg c \vee \neg d) \wedge (\neg c \vee g \vee \neg d) \wedge (\neg f \vee \neg g) \wedge (a \vee b) \wedge (\neg a \vee \neg b)$. Real expressions automatically generated while solving problems could be many thousands of clauses long.

In Prolog, the representation for the logical expressions is as a list of lists: `[ C1, C2, ... CN ]`, where the clauses `Ci` are logically ANDed together. Each `Ci` is a list of the form `[ Vi1, Vi2, ... Vi3 ]`, where the `Vij` are logically ORed together. Each `Vij` is either a Prolog atom like `a` or the negation of an atom `not(a)`. For example, the representation for the two expressions given above is `[[a,b],[not(a),b,c]]` and `[[a,not(b),c],[not(c)]]`. True and false are encoded as 1 and 0 respectively. Your algorithm should work with expressions that are arbitrarily large.

We use the standard logic tables:

| $p$ | $q$ | $p \vee q$ | $p \wedge q$ | $\neg p$ |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 | 1 |

## Requirements

The assignment requires you to develop two main and one auxiliary predicates:

- The main predicates are: `findtruth1(+Expr,-PropBind)` and `findtruth2(+Expr,-PropBind)`. The input `Expr` is a Prolog list encoding the expression. The returned `PropBind` is a list of pairs `[PropName, PropValue]`, where `PropName` is the name used for the elementary proposition and `PropValue` is the value (0 for false and 1 for true) given to that proposition. `findtruth1` and `findtruth2` find a set of truth values for each elementary proposition in `PropBind`, such that the expression `Expr` evaluates to true. One `PropBind` that satisfies both of the example expressions above is `[[a,1],[b,1],[c,0]]`.

  `findtruth1(+Expr,-PropBind)` is to solve this problem using constraint logic programming. Here the propositions take only two values, so you use expressions like "`Prop in 0..1`" to specify their values.

  `findtruth2(+Expr,-PropBind)` is to solve this problem using exhaustive generation of all possible values (*i.e.* 0 and 1) for all propositions.

  In both cases, each time you backtrack through the expression, it should generate another possible solution. I.e.,
  `((findtruth1(Expr,PropBind),print(PropBind),nl,fail);true)`
  should generate all solutions.

- You must also develop an auxiliary predicate `findvars(+Expr,-PropNames)` that scans the expression `Expr` and extracts all of the elementary proposition names. I.e., it extracts the list `[a,b,c]` for both of the example expressions above.

- Include in your program two predicates `solve1` and `solve2`, which are defined as:
  ```
  solve1(Expr) :- ((findtruth1(Expr,PropBind),print(PropBind),nl,fail);true).
  solve2(Expr) :- ((findtruth2(Expr,PropBind),print(PropBind),nl,fail);true).
  ```
  These find a solution, print it, fail to force backtracking to the next solution, and finally succeed with a `true` when backtracking doesn't find any more solutions.

## Other Information

You also need to include these statements at the start of your program:
"`:- use_module(library(clpfd)).`" and "`:- use_module(library(lists)).`" to load the constraint logic and list predicate libraries.

To run your program, you need to start up SICSTUS Prolog by typing `sicstus` on a DICE machine. Then you need to consult your solution file. (Eg. type "`[solution].`" if your solution is stored in the file "`solution.pl`").

## Some hints

1. An example solution has been done in about 50 lines of code.
2. The solution can be found in less than 1 second on a DICE machine. If your algorithm is taking much longer than this, think about the efficiency and possibly some bugs.
3. You might want to use the Sicstus manual:
   `www.sics.se/sicstus/docs/latest/html/sicstus.html/`
   and particularly the section on constrained logic programming in finite domains:
   `www.sics.se/sicstus/docs/latest/html/sicstus.html/CLPFD.html`.
4. You might use some functions from Sicstus list library. This provides many standard list manipulation predicates such as `member/2`. See
   `www.sics.se/sicstus/docs/latest/html/sicstus.html/Lists.html`
   for more details.
5. Set up some test examples to debug your support predicates before you start to debug the main predicate.
6. Use a smaller expression to debug your program before trying it with bigger ones.
7. Think recursion.

# General Instructions

You should put your solution to the tasks specified below into a single file, which should be in a format that can be loaded straight into Prolog (i.e. non-program text should be in comments). This file should be submitted electronically by **4pm, Friday 15 October**. The format of the submission line should be:

```
submit ai2 ai2ah A1 FILE
```

where `FILE` is the name of your file. I estimate it will take 15 hours work for the assignment.

Most of the tasks described here involve writing Prolog programs. It is important to realise that programs developed for any serious purpose need to be adequately documented (eg. comments, sensible variable names). This will be taken into account in the marking of your work.

The assignment will be marked by:

| Issue | Percentage |
|---|---|
| 1. Solution to `findtruth1` constraint logic algorithm | 20% |
| 2. Solution to `findtruth2` combinatorial search algorithm | 20% |
| 3. Solution to `findvars` algorithm | 20% |
| 4. Test results | 20% |
| 5. Prolog clarity | 20% |

## The Submission

For your submission, create one executable Prolog file with:

1. Your name and email address (commented out using %). Add a comment line
   `% TIME=???`
   where ??? is your estimate of the time spent on the assignment.

2. The code for `findtruth1`, `findtruth2` and `findvars` predicates and any supporting predicates.

3. The printout obtained when executing
   `solve1([[a,b],[not(a),b,c]])` and `solve2([[a,b],[not(a),b,c]])` (commented out using %).

Your file should be complete and executable as submitted without any modifications, when loaded with `[solution]` and executed with `solve1(Expr)` and `solve2(Expr)`. Marks will be deducted if the marker is unable to execute your Prolog program. Be sure to try executing your program as directly loaded into a fresh instance of Sicstus to verify that it runs properly. After we have loaded your program, we will test your code with several different expressions.

You can develop and debug your program using other Prologs but it has to run on Sicstus Prolog afterwards. This is your responsibility.

## Debugging Hints

When debugging: `spy` and `trace` are essential tools. If you want to debug a predicate `p`, say `spy(p)`. When done, say `nospy(p)`. Prolog will stop every time it gets to `p` and print `p` with its input arguments. Look at these inputs carefully. It also stops when `p` exits (ie. is finished executing). Look at the results carefully. Whenever Prolog stops, each time that you type the return key, Prolog goes to the next predicate. Thus, you can step through sections of your program.

Similarly, if you have a clause `p :- a,b,c.` and you think it's going wrong at predicate `b`, then use this: `p :- a,trace,b,c.` This will start the tracing when your program gets to `b`. As before, single step through the predicate calls to see what's happening. Review

the Prolog debugger at:
`www.sics.se/sicstus/docs/latest/html/sicstus.html/Debug-Intro.html`.

My experience with Prolog is that I need to trace through every clause at least once to help ensure that it works.

## Plagiarism

Some comments about plagiarism: 1) Submitting another student's code (even if modified) as if it is your original work is plagiarism. 2) Assisting another student to plagiarise (eg. by sharing code without the borrowing student acknowledging the use) is also penalised. 3) Discussing the assignment in broad terms is ok, but not at the level of coding details. 4) If you cannot figure out how to code some portion of your program, you **could** borrow someone else's code, **provided** you acknowledge whose code it was and label their portion of your submission clearly. You will not get credit for the other person's code. 5) Even partial, non-working submissions get partial credit. 6) A failed assignment is not a ruined career. Getting caught at plagiarism could ruin it. 7) We use various techniques to detect plagiarism, including automated tools. 8) If you can't do the assignments, discuss this with the course organiser or director of studies. 9) Change the protections on your homework files and directories so potential plagiarists cannot access your solution. If your file is called XXX, then use the unix command: `chmod go-rwx XXX`. Don't assume that your flatmates or best friends would never plagiarise from you.