AI2 Module 1 Practical 2

Bob Fisher
School of Informatics

Autumn 2004

Abstract

This document describes the second assignment (of 2) for assessment on AI2 Module 1. The main goal is to develop the resolution algorithm in class.

Task Background

Lecture 12 introduced the ideas behind the resolution algorithm as a tool for logical proof by contradiction. This assignment implements and tests the resolution algorithm.

The key input data structure is a logical expression, which (here) is 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 $(a \vee \neg b \vee c) \wedge (\neg c)$. Real expressions automatically generated while solving problems could be many thousands of clauses long.

The key step in resolution is to combine two clauses of the form $P \vee Q$ and $\neg P \vee R$ to produce $Q \vee R$. Here P, Q and R can be any logical term. So, resolving $a \vee b$ and $\neg a \vee \neg c$ produces $b \vee \neg c$. The purpose of using resolution is to derive the empty clause by resolving a with $\neg a$. Whenever this pair is resolved, then there is a contradiction and the process can stop. Otherwise, it tries to derive everything it can, and stops when no more new resolutions are possible.

When getting the resolved result $Q \vee R$, some simplifications might be needed:

1. If there are any duplicate terms in the result (*e.g.* $a \vee b \vee a$), then they should be removed (*e.g.* to $b \vee a$).

2. If a result clause contains both a and $\neg a$ (*e.g.* $a \vee b \vee \neg a$), then this result clause is a tautology, and should be not used further.

3. If a result clause is subsumed by any existing clause (*e.g.* $a \vee b \vee c$ is subsumed by $a \vee b$ because anything that makes $a \vee b$ true also makes $a \vee b \vee c$ true), then the more complex clause should be not used further. This improves the efficiency of the resolution algorithm.

You can read more about resolution in Russell and Norvig. Fortunately, this chapter is online: `aima.eecs.berkeley.edu/newchap07.pdf`.

In Prolog, the representation for the logical expressions is the same as in practical 1, 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 one main and one auxiliary predicate:

- The main predicate is `resolve(+Set))`. The input `Set` is a Prolog list encoding the set of clauses. The predicate should attempt all resolutions between each pair of clauses `Ci` and `Cj` and succeed when done, printing either "No contradiction derived" or "Contradiction derived". Note that whenever you succeed in resolving two clauses to produce a new clause, the new clause needs to be added to the Set and then also used for additional resolutions.

- You must also develop an auxiliary predicate `resolveclause(+Clause1,+Clause2,-Var,-Result))` that scans the input expressions `Clause1` and `Clause2` and resolves them if possible, returning the selected variable (`Var`) and the resulting resolved clause `Result`. Note that there might be several ways to resolve the expressions, as in $a \vee b \vee c$ and $\neg a \vee \neg b \vee d$ (which resolve to a tautology). In Prolog form, this would be resolving `C1=[a,b,c]` and `C1=[not(a),not(b),d]`. If no resolution is possible, then the predicate fails.

Other Information

You also need to include this statement at the start of your program:
"`:- use_module(library(lists)).`" to load the 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 35 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/`.
4. You might use some functions from Sicstus list library. This provides many standard list manipulation predicates that could be used in this practical, such as `member/2, select/3, permutation/2, remove_duplicates/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 5 November**. The format of the submission line should be:

```
submit ai2 ai2a A2 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 `resolve` algorithm	25%
2. Solution to `resolveclause` algorithm	25%
3. Test results	30%
4. 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 `resolve` and `resolveclause` predicates and any supporting predicates.

3. The printout obtained when executing
   `resolve([[a,not(b),c],[not(a),b,not(c)],[a],[not(b)]])` and
   `resolve([[not(p21),b11],[not(b11),p12,p21],[not(p12),b11],[not(b11)],[p12]])`.
   (commented out using %).

Your file should be complete and executable as submitted without any modifications, when loaded with `[solution]` and executed with
`resolve([[a,not(b),c],[not(a),b,not(c)],[a],[not(b)]])` and
`resolve([[not(p21),b11],[not(b11),p12,p21],[not(p12),b11],[not(b11)],[p12]])`
and other tests. 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.