

# Resolve-TP: A Resolution Theorem Prover for First Order Logic

Brian Greskamp  
netid: greskamp

April 8, 2005

## Introduction

This paper outlines `resolve-tp`, a simple resolution theorem prover for first order logic implemented in C++. `resolve-tp` accepts a knowledge base of sentences in conjunctive normal form consisting of variables, functions, constants, and predicates. It performs a refutation-complete proof procedure to determine whether the knowledge base is consistent. On top of the basic resolution procedure, `resolve-tp` implements the set-of-support and subsumption optimizations, which are discussed in the following section.

## Implementation

The program consists of a front-end and the theorem-proving back-end. The front-end is a YACC parser that reads a set of CNF sentences from a flat ASCII file and writes them into a knowledge base. The back end iterates resolution on the knowledge base until either a contradiction is found (the empty statement is produced), or there are no more sentence pairs to resolve. We will discuss the back-end only.

The back-end uses a “work list” to store pairs of sentences from the knowledge base that need to be resolved. At each iteration, the resolution engine pops the first pair of sentences off of the work list and performs resolution on them. Whenever a resolution produces a new sentence  $s$  that should be added to the knowledge base, entries of the form  $(s, x)$  are added to the worklist for all sentences  $x$  already in the knowledge base. Resolution halts when either a contradiction is found or the work list becomes empty. In other words, the work list (or queue) is used to implement breadth-first search in resolution space. Breadth-first search is required for refutation completeness. To see why, consider the knowledge base<sup>1</sup>:

- 1) `~Number(S(S(S(Zero))))`
- 2) `Number(Zero)`

---

<sup>1</sup>‘|’ stands for logical-or and ‘~’ stands for negation

3)  $\sim\text{Number}(x) \mid \text{Number}(S(x))$

If depth-first search were used, we might resolve (2, 3) to produce 4 and continue indefinitely by resolving (3,  $n$ ) to produce  $n + 1$  without ever finding the contradiction induced by 1. In contrast, the work list approach ensures that we will eventually search all pairs of sentences.

Now if we add a procedure for resolving two sentences, we have a complete (albeit inefficient) resolution procedure. To resolve two sentences  $A$  and  $B$ , we examine all pairs  $(p, q)$  of predicates  $p \in A, q \in B$  until we find one that satisfies the resolution rule. Note that this is an  $O(n^2)$  search for each resolution step. The unification algorithm described in Russel and Norvig is used to unify  $(p, q)$  and generate a binding list. Finally, a new sentence  $S$  is formed by adding all of the predicates in  $A$  and  $B$  except for  $p$  and  $q$  to  $S$  and applying all substitutions in the binding list.

Now that we have a refutation-complete engine, we endeavor to make it faster without sacrificing completeness. To this end, `resolve-tp` implements two well-known optimizations: set-of-support and subsumption. Completeness proofs will not be given here.

## Set-of-support

The order in which we search the resolution space matters. Intuitively, it would be desirable to do a backward search from the goal sentence (the sentence we are trying to contradict) rather than following an arbitrary path. To this end, define the “set of support” at some time  $t$  to be the set containing the goal sentence and all other sentences that have been produced through resolution where at least one of the parent sentences was in the set. Thus, the set of support at any given time is the set of all knowledge base sentences whose truth might depend on the truth of the goal sentence.

The set of support can be exploited as follows. Initially, the set contains only the goal sentence. The engine then iterates resolution using a worklist as before except that only sentence pairs  $(a, b)$  where  $b$  is in the set of support are added to the worklist. Whenever resolution succeeds, the newly produced sentence is added to the knowledge base and to the set of support. Worklist entries are also added for the new sentence.

## Subsumption

Often, resolution produces “redundant” sentences. Allowing these sentences to remain in the knowledge base does not threaten correctness, but it does make resolution slower. To combat this inefficiency, we avoid adding sentences to the database if they don’t give any new information and we remove old statements from the database when they are superseded by more general sentences. This process is called *subsumption* and is best demonstrated by an example. Assume the following knowledge base:

- 1) `Researcher(Kinsey) | Wrote(Kinsey, SexualBehavior} |  $\sim\text{Professor}(Kinsey)$`
- 2) `Researcher(Kinsey) | Controversial(Edmunds)`

- 3)  $\sim$ Wrote(Edmunds, SexualBehavior)
- 4) Researcher(Kinsey)

We say that sentence 4 *subsumes* sentences 1 and 2. Formally,  $a$  subsumes  $b$  iff  $a \Rightarrow b$ . When a sentence  $a$  subsumes another sentence  $b$  in the knowledge base, we can remove  $b$ . In the example, we see that no harm can come from removing sentences 1 and 2 since they are subsumed by 4. Once we know that Kinsey was a researcher, we no longer care whether Edmunds was controversial; sentence 2 does not contain any additional information.

`resolve-tp` actually computes subsumption only in a limited sense; the sentence doing the subsuming must consist of a single predicate. Given this limitation, the actual method for deciding subsumption of two sentences is straightforward given that we can compute the subsumption relation on predicates. Given a  $p$ , the only predicate in sentence  $a$  and a sentence  $b$ :

$$\text{subsumes}(a, b) \Leftrightarrow (\exists q \ q \in \text{predicates}(b) \wedge \text{subsumes}(p, q))$$

To compute the subsumption relation over predicates, a modified version of the unification algorithm is employed. Consider two predicates  $p$  and  $q$ . Traverse the arguments of each from left to right. Whenever an unbound variable is encountered in  $p$ , bind it to the corresponding expression in  $q$ . When a bound variable is encountered in  $p$ , the corresponding argument in  $q$  must exactly match the bound expression. If it does not, then we have found a conflict and  $p$  does not subsume  $q$ . If we can complete the traversal of the argument lists without encountering a conflict then  $p$  does subsume  $q$ .

## Usage

The program accepts a single command line argument: the name of the file containing the CNF knowledge base. The format of the file is straightforward as shown in the example below. Lines that begin with '#' are comments. The first non-comment line should hold the proposition to be refuted. In actuality, it can be placed anywhere in the file, but the set of support is initialized with the sentence on the first line. Variables must begin with a lower-case letter and constants must begin with an upper case letter. Function or predicate names can begin with either upper or lower case.

```
# Homework #3 Problem 2

# Proposition
 $\sim$ Buy(Tom, Cookie)

# (Better(x, y, z) & Hungry(x)) -> Buy(x, y)
 $\sim$ Better(x, y, z) |  $\sim$ Hungry(x) | Buy(x, y)

# ( $\sim$ Rich(x) & Cheap(y) & Expensive(z)) -> Better(x, y, z)
```

```

Rich(x) | ~Cheap(y) | ~Expensive(z) | Better(x, y, z)

# (Rich(x) & Cheap(y) & Expensive(z)) -> Better(x, z, y)
~Rich(x) | ~Cheap(y) | ~Expensive(z) | Better(x, z, y)

# Poor(x) -> (~Rich(x) & Hungry(x))
~Poor(x) | ~Rich(x)
~Poor(x) | Hungry(x)

# Cheap(Cookie) & Expensive(Pizza)
Cheap(Cookie)
Expensive(Pizza)

Poor(Tom)
Rich(Jim)
Hungry(Jim)

```

Running `resolve-tp` on this file yields an output trace as shown below (irrelevant sentences have been deleted from the listing). Each sentence is assigned a number (shown in the first column) when it is added to the knowledge base. The listing begins with the sentences originally in the input file, in the order they appeared. Resolved sentences follow. To the right of the number is the sentence itself, followed by a \$ symbol. After the \$ is the pair of sentences that produced it and the corresponding binding list. In the binding list, `x_1` refers to variable `x` of the first sentence in the pair and `x_2` refers to variable `x` in the second sentence.

```

0: ~Buy(Tom, Cookie)
1: ~Better(x, y, z) | ~Hungry(x) | Buy(x, y)
2: Rich(x) | ~Cheap(y) | ~Expensive(z) | Better(x, y, z)
4: ~Poor(x) | ~Rich(x)
5: ~Poor(x) | Hungry(x)
6: Cheap(Cookie)
7: Expensive(Pizza)
8: Poor(Tom)
-----
11: ~Better(Tom, Cookie, z) | ~Hungry(Tom) $ (1, 0) {x_1/Tom, y_1/Cookie}
12: ~Hungry(Tom) | Rich(Tom) | ~Cheap(Cookie) | ~Expensive(z) $ (11, 2) {x_2/Tom, y_2/Cookie, z_1/z_2}
16: ~Hungry(Tom) | ~Cheap(Cookie) | ~Expensive(z) | ~Poor(Tom) $ (12, 4) {x_2/Tom}
32: ~Cheap(Cookie) | ~Expensive(z) | ~Poor(Tom) $ (16, 5) {x_2/Tom}
118: ~Expensive(z) | ~Poor(Tom) $ (32, 6) {}
579: ~Poor(Tom) $ (118, 7) {z_1/Pizza}
589: $ (579, 8) {}
Resolution completed with 23 facts and 15 supporting

```

## Evaluation

It is not easy to disable subsumption in the implementation, so its performance impact will not be evaluated. Subjectively, however, it is likely that subsumption actually leads

to slowdown on small problems since the implementation is not very efficient. When a subsumed sentence is removed from the knowledge base, all work list entries referring to the sentence must also be removed. This could be done efficiently by maintaining a mapping from knowledge base sentence to worklist entries that contain them, but it is currently accomplished by performing a complete traversal of the work list.

The impact of set-of-support is easier to evaluate. Placing the goal sentence first in the input file (correctly initializing the set) gives a proof trace with 589 new sentences produced, 23 of which remain in the knowledge base when the contradiction is found. Placing the goal sentence at the end of the file (incorrectly initializing the set) produces a proof with over 5000 new sentences.

## Conclusion

`resolve-tp` is not particularly fast or elegant, but it does demonstrate the basic methods of resolution proof, and it is complete. Additional search heuristics and a more powerful subsumption model could greatly enhance its power.