

# **Expert Systems Knowledge Base & Inference Engine**

Jeremy Vun

## Table of Contents

1. How to Use.....	3
2. General Knowledge Base .....	3
2.1 Testing.....	4
2.2 Test cases .....	4
3. CNF solvers.....	4
3.1 Resolution .....	4
3.2 DPLL.....	5
4. Comparing DPLL and Resolution .....	5
4.1 Findings .....	5
5. References .....	6
Appendix A.....	7

## 1. How to Use

iengine <method> <filename>

Methods	
TT	Truth Table enumeration
BC	Backward chaining
FC	Forward chaining
RES	Resolution
DPLL	Davis-Putnam-Logeman-Loveland

Table 1.0 – list of available inference engine methods

## 2. General Knowledge Base

General knowledge parses a given set of propositional sentences into CNF form. It implements precedence for all propositional logic operators ( $\Leftrightarrow$ ,  $\Rightarrow$ ,  $\vee$ ,  $\wedge$ ,  $\neg$ ) and supports bracket priority.

In order to get an expression into CNF, the following steps are taken

1. Eliminate bi-directionals with conversion into implications.  $(a \Leftrightarrow b)$  gives  $((a \Rightarrow b) \wedge (b \Rightarrow a))$
2. All implications removed.  $(a \Rightarrow b)$  gives  $(\neg a \vee b)$
3. Any negations are pushed into the propositional literals.  $\neg(a \vee b)$  gives  $(\neg a \wedge \neg b)$
4. Distributive law recursively applied to split tokens into CNF clauses.  $a \vee (b \wedge c)$  gives  $(a \vee b) \wedge (a \vee c)$

```
36 vector<vector<string>> CNFKnowledgeBase::convertToCNF(string& sentence) {
37     //split sentence into tokens
38     sentence = TokenUtil::trimspaces(sentence);
39     vector<string> tokens = TokenUtil::extractTokens(sentence);
40
41     //1. eliminate <=>
42     tokens = removeBiDirection(tokens);
43
44     vector<vector<string>> clauses{ tokens };
45
46     for (int i = clauses.size() - 1; i >= 0; i--) {
47         //2. eliminate =>
48         clauses[i] = removeImplication(clauses[i]);
49         clauses[i] = TokenUtil::removeRedundantBrackets(clauses[i]);
50
51         //3. push negations in
52         clauses[i] = pushNegations(clauses[i]);
53
54         //remove unnecessary brackets by converting between infix and postfix
55         clauses[i] = TokenUtil::infixToRpn(clauses[i]);
56         clauses[i] = TokenUtil::rpnToInfix(clauses[i]);
57
58         //4. apply distributive law
59         vector<vector<string>> distributedClauses = distributive(clauses[i]);
60     }
```

Figure 2.0 – Normalisation process to get expressions into CNF

Translation between infix and postfix/reverse polish notation greatly helps evaluation and parsing of sentence symbols/tokens. The other significant benefit of translating between infix and postfix is that the process natively removes unnecessary parenthesis.

Resolution and DPLL solvers have been implemented to use the CNF general KB.

## 2.1 Testing

When any of the CNF based solvers are run, they print the KB sentences to console. KB sentences are then compared against an expected output derived from hand calculation.

## 2.2 Test cases

Input Sentences (TELL)	Expected CNF Clauses
$(a \leftrightarrow (c \rightarrow \neg d)) \ \& \ b \ \& \ (b \Rightarrow a); \ c; \ \neg f \mid g;$	$(\neg a \mid \neg c \mid \neg d); (a \mid c); (a \mid d); (b); (\neg b \mid a); (c); (\neg f \mid g)$
$((a \Rightarrow b) \mid \mid c \& d);$	$(\neg a \mid b \mid c); (\neg a \mid b \mid d)$
$\neg a \mid \neg b \Rightarrow c; \ \neg a \& \neg b \Rightarrow c; \ d \Rightarrow \neg f$	$(c \mid a); (b \mid c); (a \mid b \mid c); (\neg d \mid \neg f)$
$p2 \Rightarrow p3; \ p3 \Rightarrow p1; \ c \Rightarrow e; \ b \& e \Rightarrow f; \ f \& g \Rightarrow h;$ $p1 \Rightarrow d; \ p1 \& p3 \Rightarrow c; \ a; \ b; \ p2;$	$(a); (b); (p2); (\neg b \mid \neg e \mid f); (\neg c \mid e); (\neg f \mid \neg g \mid h);$ $(\neg p1 \mid d); (\neg p1 \mid \neg p3 \mid c); (\neg p2 \mid p3); (\neg p3 \mid p1);$
$\neg a \Rightarrow \neg b; \ \neg b \Rightarrow \neg c; \ \neg a;$	$(a \mid \neg b); (b \mid \neg c); \neg a;$
$a \leftrightarrow (b \leftrightarrow c);$	$(\neg a \mid \neg b \mid c); (\neg a \mid \neg c \mid b); (a \mid b \mid c);$ $(a \mid b \mid \neg b); (a \mid \neg c \mid c); (a \mid \neg c \mid \neg b);$
$((a \mid b) \& (\neg a \mid c)) \mid \mid d;$	$(d \mid a \mid b); (d \mid \neg a \mid c)$
$z; \ (y \mid a); \ p \leftrightarrow (g \& h);$	$(y \mid a); (z); (\neg a); (\neg g \mid \neg h \mid p); (\neg p \mid g); (\neg p \mid h)$
$(a \& b) \mid \mid (x \& y);$	$(a \mid x); (a \mid y); (b \mid x); (b \mid y)$
$((a \mid b) \& c);$	$(a \mid b); (c)$
<none>	<none>

## 3. CNF solvers

### 3.1 Resolution

The resolution solver throws the negated TELL query into the KB and combines CNF clauses together until either an empty clause is produced (query is satisfied by negation), or no new clauses can be created (query is unsatisfiable).

In order to make sure the resolution solver completes, duplicate symbols in resolvent products are removed otherwise the resolution algorithm can end up looping forever.











▶  ["a a a ~d"]	false
▶  ["a a a ~d ~d"]	false
▶  ["a a a ~d ~d ~d"]	false
▶  ["a a a ~d ~d ~d ~d"]	false
▶  ["a a a ~d ~d ~d ~d ~d"]	false
▶  ["a a a ~d ~d ~d ~d ~d ~d"]	false
▶  ["a a a ~d ~d ~d ~d ~d ~d ~d"]	false
▶  ["a a a ~d ~d ~d ~d ~d ~d ~d ~d"]	false
▶  ["a a a ~d ~d ~d ~d ~d ~d ~d ~d ~d"]	false
▶  ["a a c"]	false

Figure 3.0 – Example of endless term explosion

Hash tables are used to keep track of resolvents already seen to linearise checking time of resolution products and reduce memory consumption storing duplicates.

### 3.2 DPLL

Based on resolution, Davis-Putnam-Logemann-Loveland algorithm reduces memory usage and speeds up the SAT problem primarily through unit propagation of symbol literals.

1. Search for unit clauses in the CNF expression and propagate them.

For example, given the following KB, where  $\sim e$  is pushed into the KB to prove by negation.

**KB<sub>0</sub>:**  $a // b; \sim a // e; a; \sim e;$

Because  $a$  is known to be true, all clauses containing  $a$  can be simplified, giving:

**KB<sub>1</sub>:**  $e; \sim e$

$\sim e$  can be propagated next, producing an empty clause  $\{ \}$  and indicating that the KB entails the original query  $e$ .

1. Where no unit literals can be propagated, DPLL chooses a symbol at random and recursively branches true/false on it.
2. If an empty clause is produced, entailment is proved by negation. Otherwise, if there are no more symbols to propagate, then the query is not entailed from the KB.

## 4. Comparing DPLL and Resolution

Results of the relative performance of DPLL, Resolution, and Truth Table Enumeration over a number of propositional formula. The time taken to populate the KB is not included - only the inference engine query time is measured. Tests conducted on the same machine.

		Resolution (ms)	DPLL (ms)	Truth Table (ms)
<b>Formula 1</b>	<b>TELL:</b> $(a \Leftrightarrow (c \Rightarrow \sim d)) \ \& \ b \ \& \ (b \Rightarrow a); \ c; \ \sim f \mid g;$ <b>ASK:</b> $a;$	2.86	1.75	0.41
<b>Formula 2</b>	<b>TELL:</b> $p2 \Rightarrow p3; \ p3 \Rightarrow p1; \ c \Rightarrow e; \ b \ \& \ e \Rightarrow f; \ f \ \& \ g \Rightarrow h; \ p1 \Rightarrow d;$ $p1 \ \& \ p3 \Rightarrow c; \ a; \ b; \ p2;$ <b>ASK:</b> $d;$	6.39	2.47	3.98
<b>Formula 3</b>	<b>TELL:</b> $a \Leftrightarrow (b \Leftrightarrow c); \ d \Leftrightarrow (e \Leftrightarrow f); \ g \Leftrightarrow (h \Leftrightarrow i);$ $j \Leftrightarrow (k \Leftrightarrow l);$ <b>ASK:</b> $h;$	30.51	5.08	40.02

Table 4.0 – each formula run 10 times and average time recorded

### 4.1 Findings

- DPLL outperforms Truth table enumeration and Resolution. It also scales significantly better with the size of the SAT problem.
- With smaller/easier expressions, Truth table enumeration is faster due to lower overhead.
- However, Truth table performance decreases rapidly as the number of symbols increases due to the exponential time complexity of truth table enumeration.
- Resolution has a lot of overhead associated with resolving clauses and decreases in performance as the number of clauses in the CNF expression increases. Formula 3 produces a lot of clauses.

## 5. References

Russel, S. & Norvig, P 2009, Artificial Intelligence: A Modern Approach, 3<sup>rd</sup> edition, Prentice Hall Press

## Appendix A

	Formula 1			Formula 2			Formula 3		
Runs	RES	DPLL	TT	RES	DPLL	TT	RES	DPLL	TT
1	2.932	1.7966	0.3974	6.1953	2.428	3.78	29.7736	4.9244	39.1145
2	2.6095	2.0839	0.4208	6.1733	2.4187	3.6733	30.5283	5.1792	40.7183
3	2.6885	1.567	0.4129	6.2611	2.4292	4.1139	30.5231	5.8493	38.2075
4	2.6363	1.5648	0.4261	6.2103	2.5165	4.073	30.8549	4.8749	40.2235
5	3.0165	1.8175	0.4408	6.3086	2.5726	4.6335	29.7722	4.8507	39.3437
6	3.0856	1.6245	0.4317	6.1866	2.426	4.5207	30.102	5.0022	40.6421
7	2.8155	1.7858	0.3819	6.2039	2.4312	3.5978	31.5121	4.9615	40.6218
8	3.0226	1.6323	0.4044	6.29	2.4125	3.8966	30.2512	5.0364	41.5237
9	3.0891	1.6217	0.398	6.3503	2.5491	3.7455	31.095	4.9425	39.4712
10	2.684	1.973	0.4307	7.7237	2.4952	3.732	30.6637	5.1567	40.3303
Average	2.85796	1.74671	0.41447	6.39031	2.4679	3.97663	30.50761	5.07778	40.01966