Conjunctive Normal Form

Bill Noble
Tim van der Molen

January 2014

1 Introduction

A formula is in conjunctive normal form (CNF) if it is a conjunction of subformulas, each of which either is a literal (i.e. an atom or the negation of an atom) or a disjunction of literals. As a special case, a CNF formula may consist of only one conjunct, in which case it simply is either a literal or a disjunction of literals. CNF is very useful in automatic theorem proving, because its structure allows for efficient formula evaluation. In this report we discuss and compare two algorithms to transform arbitrary propositional formulas into CNF. The first algorithm employs syntactical manipulation while the second has a semantical approach relying on truth tables. The two algorithms have been implemented in the Python programming language.

2 Formula Representation

We use tuples and lists to represent formulas in Python. As may be expected, tuples in Python are enclosed in round brackets while lists are enclosed in square brackets. The elements of a tuple or a list are separated by commas. For example, `(a, b, c)` is a tuple and `[a, b, c]` is a list. More specifically, a formula representation is recursively defined as follows.

1. If `s` is a string, then `(s,)` is a formula.
2. If `f` is a formula, then `('not', f)` is a formula.
3. If `f` and `g` are formulas, then `('arrow', f, g)` is a formula.
4. If $\mathtt{f}_1, \ldots, \mathtt{f}_n$ are formulas, then $\mathtt{('and', [f}_1\mathtt{, \ldots, f}_n\mathtt{])}$ is a formula.
5. If $\mathtt{f}_1, \ldots, \mathtt{f}_n$ are formulas, then $\mathtt{('or', [f}_1\mathtt{, \ldots, f}_n\mathtt{])}$ is a formula.

For example, the formula $p \wedge \neg q \wedge (r \rightarrow s)$ is represented as `('and', [('p',), ('not', ('q',)), ('arrow', ('r',), ('s',))])`.

3 The Syntactic Algorithm

The syntactic CNF algorithm exploits the distributivity of disjunction over conjunction to transform formulas into CNF. For example, the non-CNF formula $\varphi \vee (\psi \wedge \chi)$ may be transformed into the logically equivalent CNF formula $(\varphi \vee \psi) \wedge (\varphi \vee \chi)$ by distributing disjunctions over conjunction.

The syntactic CNF algorithm is applicable only to formulas that are in negation normal form (NNF). A formula is in NNF if only atoms are negated and the only other logical operators that occur in it are disjunction and conjunction. Therefore, in order for a formula to be transformed into CNF, it first must have been transformed into NNF.

The NNF algorithm, in turn, is applicable only to formulas in which negation, disjunction and conjunction occur as the only operators. Hence, every implicative subformula of the form $\varphi \rightarrow \psi$ must be transformed into an equivalent disjunction of the form $\neg\varphi \vee \psi$. This is done by the impl_to_disj() function, which may be defined as follows.

1. If φ is an atom, then $\text{impl\_to\_disj}(\varphi) = \varphi$.
2. If $\varphi = \neg\psi$, then $\text{impl\_to\_disj}(\varphi) = \neg\text{impl\_to\_disj}(\psi)$.
3. If $\varphi = \psi \to \chi$, then $\text{impl\_to\_disj}(\varphi) = \neg\text{impl\_to\_disj}(\psi) \vee \text{impl\_to\_disj}(\chi)$.
4. If $\varphi = \psi_1 \wedge \ldots \wedge \psi_n$, then $\text{impl\_to\_disj}(\varphi) = \text{impl\_to\_disj}(\psi_1) \wedge \ldots \wedge \text{impl\_to\_disj}(\psi_n)$.
5. If $\varphi = \psi_1 \vee \ldots \vee \psi_n$, then $\text{impl\_to\_disj}(\varphi) = \text{impl\_to\_disj}(\psi_1) \vee \ldots \vee \text{impl\_to\_disj}(\psi_n)$.

The Python implementation is as follows.

```
def impl_to_disj(f):
    if atom(f):
        return f
    if f[0] == 'not':
        return ('not', impl_to_disj(f[1]))
    if f[0] == 'or' or f[0] == 'and':
        return (f[0], [impl_to_disj(g) for g in f[1]])
    if f[0] == 'arrow':
        return ('or', [('not', impl_to_disj(f[1])), impl_to_disj(f[2])])
    else:
        raise ValueError('unknown operator:', f[0])
```

The atom() function returns true if its tuple argument contains exactly 1 argument and false otherwise:

```
def atom(f):
    return len(f) == 1
```

The nnf() function may be defined as follows.

1. If φ is an atom, then $\text{nnf}(\varphi) = \varphi$.
2. If $\varphi = \neg\psi$, then:
    a) If ψ is an atom, then $\text{nnf}(\varphi) = \varphi$.
    b) If $\psi = \neg\chi$, then $\text{nnf}(\varphi) = \text{nnf}(\chi)$.
    c) If $\psi = \chi_1 \wedge \ldots \wedge \chi_n$, then $\text{nnf}(\varphi) = \text{nnf}(\neg\chi_1) \vee \ldots \vee \text{nnf}(\neg\chi_n)$.
    d) If $\psi = \chi_1 \vee \ldots \vee \chi_n$, then $\text{nnf}(\varphi) = \text{nnf}(\neg\chi_1) \wedge \ldots \wedge \text{nnf}(\neg\chi_n)$.
3. If $\varphi = \psi_1 \wedge \ldots \wedge \psi_n$, then $\text{nnf}(\varphi) = \text{nnf}(\psi_1) \wedge \ldots \wedge \text{nnf}(\psi_n)$.
4. If $\varphi = \psi_1 \vee \ldots \vee \psi_n$, then $\text{nnf}(\varphi) = \text{nnf}(\psi_1) \vee \ldots \vee \text{nnf}(\psi_n)$.

The Python implementation is as follows.

```
def nnf(f):
    return nnf_do(impl_to_disj(f))

def nnf_do(f):
    if atom(f):
        return f
    if f[0] == 'not':
```

```
        if atom(f[1]):
            return f
        if f[1][0] == 'not':
            return nnf_do(f[1][1])
        if f[1][0] == 'and':
            return ('or', [nnf_do(('not', g)) for g in f[1][1]])
        if f[1][0] == 'or':
            return ('and', [nnf_do(('not', g)) for g in f[1][1]])
        else:
            raise ValueError('unexpected operator:', f[1][0])
    if f[0] == 'and' or f[0] == 'or':
        return (f[0], [nnf_do(g) for g in f[1]])
    else:
        raise ValueError('unexpected operator:', f[0])
```

The cnf() function is defined as follows.

1. If φ is a literal, then $\mathrm{cnf}(\varphi) = \varphi$.
2. If $\varphi = \psi_1 \wedge \ldots \wedge \psi_n$, then $\mathrm{cnf}(\varphi) = \mathrm{cnf}(\psi_1) \wedge \ldots \wedge \mathrm{cnf}(\psi_n)$.
3. If $\varphi = \psi_1 \vee \ldots \vee \psi_n$, then $\mathrm{cnf}(\varphi) = \mathrm{dist}(\mathrm{cnf}(\psi_1), \mathrm{cnf}(\psi_2 \vee \ldots \vee \psi_n))$.

The dist() function performs the disjunction distribution and is defined as follows.

1. If $\varphi = \chi_1 \wedge \ldots \wedge \chi_n$, then $\mathrm{dist}(\varphi, \psi) = \mathrm{dist}(\chi_1, \psi) \wedge \ldots \wedge \mathrm{dist}(\chi_n, \psi)$.
2. If $\psi = \chi_1 \wedge \ldots \wedge \chi_n$, then $\mathrm{dist}(\varphi, \psi) = \mathrm{dist}(\varphi, \chi_1) \wedge \ldots \wedge \mathrm{dist}(\varphi, \chi_n)$.
3. Otherwise, $\mathrm{dist}(\varphi, \psi) = \varphi \vee \psi$.

The algorithm may be described as follows. It starts at the deepest nested conjunctions and, if necessary, applies the distribution rule. This moves the original conjunctions up one level in the formula. At this level distribution is applied again, if necessary. This process is continued until all conjunctions have been pushed up above any disjunction.

The Python implementation of cnf() and dist() is as follows.

```
def cnf(f):
    if atom(f) or f[0] == 'not':
        return f
    if f[0] == 'and':
        return ('and', [cnf(g) for g in f[1]])
    if f[0] == 'or':
        if len(f[1]) == 0:
            return f
        if len(f[1]) == 1:
            return cnf(f[1][0])
        else:
            return dist(cnf(f[1][0]), cnf(('or', f[1][1:])))
    else:
        raise ValueError('unknown operator:', f[0])

def dist(f, g):
    if f[0] == 'and':
        if len(f[1]) == 0:
```

```
            return f
        else:
            return ('and', [dist(h, g) for h in f[1]])
    if g[0] == 'and':
        if len(g[1]) == 0:
            return g
        else:
            return ('and', [dist(f, h) for h in g[1]])
    else:
        return ('or', [f, g])
```

After a formula has been transformed into CNF, it may contain nested lists of conjunctions and disjunctions. These lists are flattened or collapsed by the flatten() function. As a final step, the remove_dups() function removes from every disjunction list duplicate disjuncts and likewise for the conjunction list.

The Python implementation for these functions is as follows.

```
def flatten(f):
    if f[0] == 'and':
        return ('and', flatten_conj(f[1]))
    if f[0] == 'or':
        return ('or', flatten_disj(f[1]))
    else:
        return f

def flatten_conj(flist):
    if flist == []:
        return flist
    if flist[0][0] == 'and':
        return flatten_conj(flist[0][1] + flist[1:])
    else:
        return [flatten(flist[0])] + flatten_conj(flist[1:])

def flatten_disj(flist):
    if flist == []:
        return flist
    if flist[0][0] == 'or':
        return flatten_disj(flist[0][1] + flist[1:])
    else:
        return [flist[0]] + flatten_disj(flist[1:])

def remove_dups(f):
    if f[0] == 'and':
        flist = []
        for g in f[1]:
            g = remove_dups(g)
            if not g in flist:
                flist.append(g)
        return ('and', flist)
    if f[0] == 'or':
        return ('or', list(set(f[1])))
    else:
        return f
```

To incorporate these functions into the function chain, the original cnf() function is renamed to cnf_do() and the new cnf() function is as follows.

```
def cnf(f):
    return remove_dups(flatten(cnf_do(nnf(f))))
```

4 The Semantic Algorithm

Rather than converting f to CNF directly, the cnf_tt algorithm constructs a truth table and uses it to produce a formula in CNF that is logically equivalent to f (i.e. it is true at exactly the rows where f is true). It is more intuitive to use this strategy to produce a formula in DNF.

p	q	r	$(p \vee q) \to r$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

To write a formula that is true at exactly one row in a truth table (i.e. it is true there and nowhere else), one takes the conjunction of all the literals or their negations depending on their truth or falsity at that row. For example, the formula $\wedge[\neg p, q, r]$ is true at exactly row 4 in the truth table above. Reading a DNF formula from a truth table is an extension of this idea: just take the disjunction of such formulas for each row where f is true. The resulting formula is in DNF and is true at exactly the lines where f is true (i.e. it is logically equivalent). Thus we have:

$$\vee[p, q] \to r \equiv \vee[\wedge[\neg p, \neg q, \neg r], \wedge[\neg p, \neg q, r], \wedge[\neg p, q, r], \wedge[p, \neg q, r], \wedge[p, q, r]]$$

```
# Uses the truth table method to compute disjunctive normal form
def dnf_tt(f):
    # filter the truth table to only rows where f is True
    f_true_tt = [row for row in gen_tt(get_atoms(f))
                 if evaluate(f, row) == True]
    # return an 'or' list of 'and' lists (one for each row in
    # f_true_tt) such that each 'and' list contains p when
    # p is True in the row and ('not', p) when it is False.
    return tuple(['or', [tuple(['and',
        [p if row[p] else tuple(['not', p]) for p in row.keys() ]])
        for row in f_true_tt] ])
```

To read the CNF from the truth table, we take the opposite (but obviously logically equivalent) approach: the goal is to write a formula that is false at exactly the lines where f is false. Say we wanted to write a formula that is false at exactly line 3. The formula $\wedge[\neg p, q, \neg r]$ is true at exactly line 3, so its negation is false exactly there. Applying the De Morgan's rule gives $\vee[p, \neg q, r]$. Naturally, taking the conjunction of such formulas, we can pick out more rows. Thus we have:

$$\vee[p, q] \to r \equiv \wedge[\vee[p, \neg q, r], \vee[\neg p, q, r], \vee[\neg p, \neg q, r],]$$

```
# Uses the truth table method to compute conjunctive normal form
def cnf_tt(f):
    # filter the truth table to only rows where f is False
```

```
    f_false_tt = [row for row in gen_tt(get_atoms(f))
                  if evaluate(f, row) == False]
    # return an 'and' list of 'or' lists (one for each row in
    # f_false_tt) such that each 'or' list contains ('not', p) when
    # p is True in the row and p when it is False.
    return tuple(['and', [tuple(['or',
        [p if not row[p] else tuple(['not', p]) for p in row.keys()] ])
        for row in f_false_tt]])
```

5 Analysis

The *cnf_tt* algorithm has a higher up-front cost than *cnf*. The cost of generating the empty truth table is relatively small; It is a list of the 2^n possible valuations (where n is the number of atomic propositions):

```
# Generates all possible valuations for a given set of atoms.
def gen_tt(atoms):
    from itertools import product
    return [{p:val for (p, val) in zip(atoms, vals)} for vals in
          product([False, True], repeat=len(atoms))]
```

Filtering the truth table to only the rows where f is $False$ is rather intensive since the formula must be evaluated at each of the 2^n rows and the evaluation function may be called up to once for each subformula of f. Nevertheless, *cnf_tt* performs better on longer random formulas, especially where total number of propositions greatly exceeds the total number of unique propositions.

Below: time difference (*cnf* minus *cnf_tt*) in converting 5000 random formulas to CNF. The x axis ranges over the number of unique propositions, and the y axis is the total number of propositions in the formula.

	1	2	3	4	5	6	7	8	9	10
1	-0.03	-0.04	-0.04	-0.04	-0.04	-0.04	-0.04	-0.04	-0.04	-0.04
2	-0.02	-0.04	-0.04	-0.04	-0.04	-0.05	-0.05	-0.05	-0.05	-0.05
3	-0.01	-0.03	-0.04	-0.06	-0.07	-0.07	-0.08	-0.08	-0.08	-0.08
4	0.01	-0.02	-0.05	-0.07	-0.10	-0.12	-0.13	-0.14	-0.15	-0.15
5	0.04	0.00	-0.04	-0.09	-0.14	-0.18	-0.22	-0.25	-0.27	-0.29
6	0.06	0.03	-0.04	-0.11	-0.19	-0.27	-0.35	-0.41	-0.47	-0.51
7	0.09	0.06	-0.02	-0.11	-0.23	-0.36	-0.50	-0.63	-0.73	-0.84
8	0.13	0.08	0.01	-0.10	-0.27	-1.06	-1.12	-1.08	-1.12	-1.80
9	0.15	0.12	0.06	-0.10	-0.29	-0.56	-0.85	-1.19	-1.53	-1.88
10	0.21	0.16	0.08	-0.07	-0.31	-0.62	-1.03	-1.57	-2.08	-2.63
11	0.26	0.22	0.13	-0.05	-0.31	-0.72	-1.27	-1.93	-2.61	-3.61
12	0.31	0.27	0.17	0.01	-0.28	-0.76	-1.43	-2.20	-3.30	-4.54
13	0.37	0.33	0.24	0.06	-0.26	-0.80	-1.57	-2.56	-4.06	-6.04
14	0.44	0.40	0.30	0.10	-0.22	-0.80	-1.73	-2.98	-4.53	-6.76
15	0.52	0.47	0.38	0.18	-0.13	-0.82	-1.84	-3.26	-5.33	-8.64
16	0.84	0.56	0.47	0.28	-0.11	-0.80	-1.86	-3.45	-6.41	-10.14
17	0.72	0.64	0.56	0.36	-0.03	-0.75	-1.93	-3.77	-6.70	-11.17
18	0.81	0.76	0.66	0.44	0.09	-0.70	-2.02	-4.02	-7.13	-11.94
19	0.96	0.88	0.80	0.61	0.19	-0.64	-1.95	-4.39	-7.89	-13.46
20	1.09	1.02	0.89	0.73	0.29	-0.52	-1.83	-4.76	-8.39	-15.43
21	1.47	1.18	1.09	0.87	0.44	-0.33	-1.87	-4.45	-8.76	-16.90
22	1.41	1.31	1.23	1.03	0.66	-0.29	-1.75	-4.55	-9.13	-17.70
23	1.61	1.54	1.41	1.25	0.80	-0.11	-1.77	-4.51	-9.48	-19.67
24	1.79	1.71	1.53	1.48	0.96	0.25	-1.54	-4.14	-10.33	-19.40
25	2.12	1.98	1.98	1.69	1.14	0.43	-1.42	-4.72	-12.29	-24.59

For formulas given in DNF, *cnf_tt* is faster for formulas of any length sentences, and *cnf* reaches Python's default maximum recursion depth of 1000 beginning at 3 propositions.

The run time of *cnf* is agnostic to the number of unique propositions in the given formula since uniqueness of a proposition is primarily semantic notion. The semantic algorithm, *cnf_tt*, is less concerned with the syntactic structure of the given formula since this has less bearing on the construction of a truth table. The relative strengths of the two algorithms highlights two distinct ways that a propositional formula can be complex.