Automated Theorem Proving in Python

Bill Noble
Tim van der Molen

June 2014

1 Introduction

The use of computer programs to prove logical theorems is called automated theorem proving. Over time, various proof procedures have been developed. Two of these are semantic tableaux and resolution. In this report, we shall discuss these two proof procedures and our implementation of them in the Python programming language.

2 Proof Procedures for Propositional Logic

We shall discuss two proof procedures: resolution and semantic tableaux. They are so-called refutation systems. That is, to prove a formula φ, the procedures attempt to refute $\neg\varphi$.

2.1 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 $\texttt{f}_1, \ldots, \texttt{f}_n$ are formulas, then $\texttt{('and', [f}_1\texttt{, ..., f}_n\texttt{])}$ is a formula.
5. If $\texttt{f}_1, \ldots, \texttt{f}_n$ are formulas, then $\texttt{('or', [f}_1\texttt{, ..., f}_n\texttt{])}$ 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',))])`.

2.2 Conjunctive Normal Form

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. We shall discuss and compare two algorithms to transform arbitrary propositional formulas into CNF. The first algorithm employs syntactical manipulation while the second has a semantic approach relying on truth tables.

2.2.1 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 $\varphi \to \psi$ 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.

```python
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:

```python
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 $\mathrm{nnf}(\varphi) = \mathrm{nnf}(\psi_1) \vee \ldots \vee \mathrm{nnf}(\psi_n)$.

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))))
```

2.2.2 The Semantic Algorithm

Rather than converting f to CNF directly, the semantic 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). An analogous and more intuitive strategy exists to produce a formula in disjunctive normal form (DNF). For example, suppose one would like to transform the formula $(p \vee q \rightarrow q) \rightarrow r$ into DNF. Consider the truth table of this formula:

p	q	r	$(p \vee q) \rightarrow r$
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 atoms 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. Constructing 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:

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

The Python implementation is as follows.

```
# 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 construct a CNF formula 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]])
```

2.2.3 Comparison

The semantic algorithm has a higher up-front cost than the syntactic one. The cost of generating the empty truth table is depends on the number of atomic propositions since 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, the semantic algorithm performs better on longer random formulas, especially where total number of propositions greatly exceeds the total number of unique propositions.

Figure 1 gives the time differences (the execution time of the syntactic algorithm minus that of the semantic one) when converting 5000 random formulas to CNF. The columns represent the number of unique atoms in a formula while the rows represent the total number of atoms occurring in that formula.

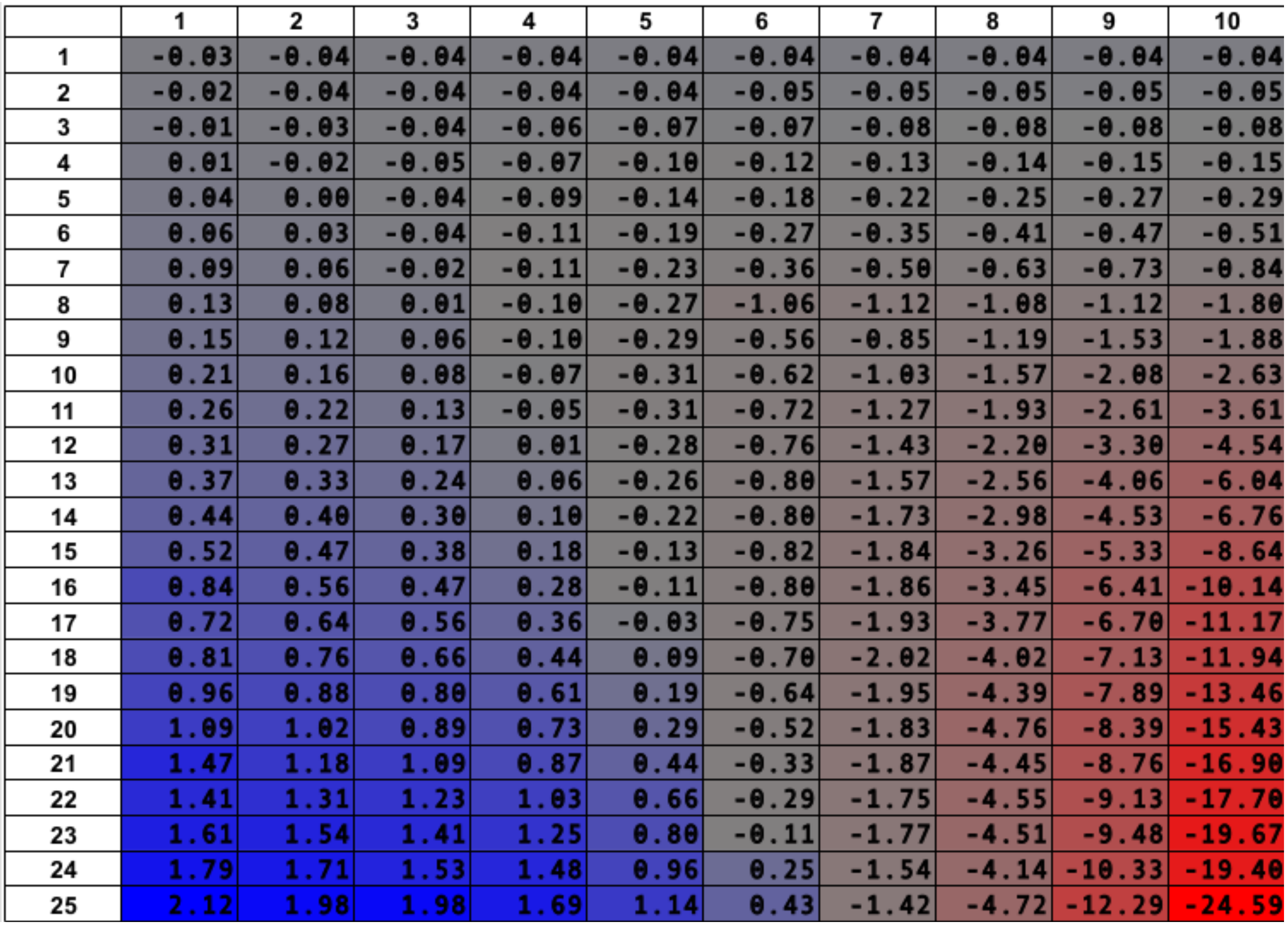
For formulas given in DNF, the semantic algorithm is faster for formulas of any length, and the syntactic algorithm reaches Python's default maximum recursion depth of 1000 beginning at 3 propositions.

The run time of the syntactic algorithm is agnostic to the number of unique propositions in the given formula since uniqueness of a proposition is primarily semantic notion. The semantic algorithm 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.

2.3 Resolution

Resolution is a proof method commonly used in automated theorem pr overs.

Figure 1: Semantic vs. syntactic CNF time differential



	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

2.3.1 Clausal Form

Before implementing resolution, it is useful to have another way to represent propositional formulas. We shall refer to a list of literals as a *clause*. A clause is satisfied if any of its literals are satisfied. Thus a non-empty clause is equivalent to a disjunction of its members. Note that an empty clause is unsatisfiable. We shall say that a list of clauses is satisfied if all of clauses in the list are satisfied. Clearly a list of clauses containing an empty clause is unsatisfiable. A list of non-empty clauses is equivalent to a formula in CNF.

We define the following utility function for converting a formula in CNF to a list of clauses (i.e., a list of lists of literals).

```
def fml_to_clauses(f):
    if atom(f) or f[0] == 'not':
        return [[f]]
    if f[0] == 'or':
        return [f[1]]
    if f[0] == 'and':
        return [fml_to_clauses(x)[0] for x in f[1]]
```

2.3.2 The Resolution Rule

The rule at the core of resolution is as follows: Given two clauses:

$$A = [a_1, \ldots, a_{i-1}, p, a_i, \ldots, a_n] \text{ and}$$
$$B = [b_1, \ldots, b_{j-1}, \neg p, b_j, \ldots, b_m]$$

such that each a_i and b_i are literals and p is an atomic formula, if A and B are both satisfied, then so is $C = [a_1, ..., a_n, b_1 ..., b_m]$. Two clauses of the form A and B are called *resolvable*. We apply the resolution rule to a list of clauses by removing two resolvable clauses and replacing them with the result of resolving them. Repeated application of the resolution rule will result in a list of clauses that are pairwise unresolvable. If $[A_1, ..., A_n]$ is a non-empty list of unresolvable clauses, it is easy to see that it is satisfiable, since we may simply choose one literal from each clause to make true, and doing so won't preclude the satisfiability of any of the other clauses. If any of the clauses is empty then the list is contradictory. This is natural since the resolution rule can only give an empty clause is by resolving two contradictory singleton clauses:

$$\left[A_1, ..., A_{i-1}, [p], [\neg p], A_i, ..., A_n\right] \overset{resolve}{\Longrightarrow} \left[A_1, ..., A_{i-1}, [\,], A_i, ..., A_n\right]$$

Clearly once an empty clause produced, there is no need to resolve further; the list is unsatisfiable.

2.3.3 Implementation

The deduction theorem for propositional logic says that a sentence ψ follows from a finite list of sentences $\varphi_1, ..., \varphi_n$ if and only if the implication formed by the conjunction of the φ_i's and ψ is a tautology:

$$\varphi_1, ..., \varphi_n \models \psi \iff \models \varphi_1 \wedge, ..., \wedge \varphi_n \to \psi$$

A sentence is a tautology if and only if its negation is a contradiction. The resolution rule can tell us whether a list of clauses is a contradiction. Thus the problem of checking if $\varphi_1, ..., \varphi_n \models \psi$ is reduced to checking whether or not a single sentence is contradictory.

```python
def resolve(f):
    return resolve_do(fml_to_clauses(cnf(('not', f))))
```

Thus `resolve()` shall return `True` when it finds contradictory clauses and `False` otherwise.

First we define a function that, given a list of clauses, searches for two that are resolvable and returns a triple containing the two clauses and the literal that makes them resolvable.

```python
def resolve_resolvable(clauses):
    for c1 in clauses:
        for lit in c1:
            if atom(lit):
                for c2 in clauses:
                    if c2 is c1:
                        continue
                    if ('not', lit) in c2:
                        return (lit, c1, c2)
```

If no resolvable clauses are found, the function will return `None` (Python's default return value).

Now we are ready to define the resolution algorithm:

```python
def resolve_do(clauses):

    res = resolve_resolvable(clauses)
    if not res:
        # No resolvable clauses were found.
        return False
    lit, c1, c2 = res

    # Apply the resolution rule.
    clauses.remove(c1)
    c1.remove(lit)
    clauses.remove(c2)
    c2.remove(('not',lit))
    clauses.append(c1 + c2)

    if [] in clauses:
        # The list of clauses is unsatisfiable.
        return True

    return resolve_do(clauses)
```

2.4 Semantic Tableaux

The semantic tableaux procedure is closely related to disjunctive normal form (DNF). Analogous to CNF, a formula is in DNF if it is a disjunction of subformulas, each which either is a literal or a conjunction of literals. As a special case, a DNF formula may consist of only a single disjunct that is either a literal or a conjunction of literals. We have implemented a syntactic DNF algorithm that is symmetric to our implementation of the syntactic CNF algorithm. Because of this symmetry, we omit the DNF implementation here.

To prove a formula φ, the semantic tableaux procedure begins by taking $\neg\varphi$ and then repeatedly applying so-called expansion rules to it. These expansion rules expand $\neg\varphi$ into a tableau or tree with the following properties. Each node is labelled by a subformula of $\neg\varphi$. Each branch represents the conjunction of its nodes, while the tree itself represents the disjunction of its branches. If the expansion is completed, the tree represents a DNF formula equivalent to $\neg\varphi$.

The expansion rules work as follows. Given a tree, take a branch and a formula ψ on that branch. Then one of the following rules are applied.

- If ψ is of the form $\neg\neg\chi$, then add χ as a new node to the branch.
- If ψ is an alpha (i.e. conjunctive) formula, then add each conjunct of ψ as a new node to the branch.
- If ψ is a beta (i.e. disjunctive) formula, then add each disjunct as a new child to the last node of the branch.

We say that a branch is closed if it contains both a formula and its negation. A tableau is closed if all of its branches are closed. Thus φ is proved if the tableau for $\neg\varphi$ is closed.

2.4.1 A Simple Semantic Tableaux Implementation

The approach outlined above suggests a very simple semantic tableaux implementation. We first call the `dnf()` function to transform the formula into DNF. This will provide us with a finished tableau. We then only have to check if all branches are closed.

```
def tableau_dnf(f):
    return tableau_dnf_do(dnf(('not', f)))

def tableau_dnf_do(f):
    if literal(f):
        return False
    if f[0] == 'and':
        return any(atom(g) and ('not', g) in f[1] for g in f[1])
    else:
        return all(tableau_dnf_do(g) for g in f[1])
```

The drawback of this approach is that it requires the DNF transformation to be completed before we can check for closure. It would be much more efficient if we perform closure tests at earlier stages in the DNF transformation. This is what we will look at now.

2.4.2 A Better Tableaux Implementation

Instead of using the `dnf()` function, we perform the expansion rules described above. But before applying an expansion rule on a branch, we check if the branch is closed. If it is, no further expansion of the branch is necessary. We use the following function to check whether a branch is closed.

```
def tableau_closed(branch):
    negations = [f for f in branch if f[0] == 'not']
    for f in negations:
        if f[1] in branch:
            return True
    return False
```

Now, the expansion rules are applied recursively by the following function. It takes a single branch as argument, applies one expansion rule to a formula on the branch and then calls itself with the resulting branch. If the expansion rule results in multiple branches, it calls itself separately for each branch.

```
def tableau_do(branch):
    if tableau_closed(branch):
        return True

    # Handle double negations and alpha formulas.
    for f in branch:
        if f[0] == 'not' and f[1][0] == 'not':
            return tableau_do([g for g in branch if g != f] + [f[1][1]])
        if f[0] == 'and':
            return tableau_do([g for g in branch if g != f] + f[1])
        if (f[0] == 'not' and f[1][0] == 'or'):
            return tableau_do([g for g in branch if g != f] +
                [('not', g) for g in f[1][1]])
        if (f[0] == 'not' and f[1][0] == 'arrow'):
            return tableau_do([g for g in branch if g != f] +
                [f[1][1], ('not', f[1][2])])

    # Handle beta formulas.
    for f in branch:
        if f[0] == 'or':
            return all(tableau_do([h for h in branch if h != f] + [g])
                for g in f[1])
        if f[0] == 'not' and f[1][0] == 'and':
            return all(tableau_do([h for h in branch if h != f] + [('not', g)])
                for g in f[1][1])
        if f[0] == 'arrow':
            return (tableau_do([g for g in branch if g != f] + [('not', f[1])])
                and tableau_do([g for g in branch if g != f] + [f[2]]))

    return False
```

Finally, the tableaux procedure is initiated by negating the formula to be proved, putting it on a branch by itself and passing it to `tableau_do()`:

```
def tableau(f):
    return tableau_do([('not', f)])
```

3 Predicate Methods

3.1 Formula Representation

In propositional logic, atomic formulas are structureless; only their names matter. Thus we were free to represent propositions simply as strings of characters.

In predicate logic, we also want to talk about relations between and functions on individuals. A *term* refers (rigidly or as a variable) to a single individual.

Formally, terms are defined inductively:

1. If `s` is a string beginning with 'w', 'x', 'y' or 'z', then `s` is a variable. A variable is a term.
2. If `s` is a string beginning with 'a', 'b', 'c', 'd', 'e', 'f' or 'g', then `s` is a function symbol.
3. If `s` is a function symbol then `(s, [])` is a constant. A constant is a term.
4. If `s` is a n-ary function symbol, and `[`$t_1,\ldots,t_n$`]` is a list of terms, then `(s,[`$t_1,\ldots,t_n$`])` is a term.

The functions for determining if a term is a variable, a contstant, or a function are defined as follows:

```
def variable(x):
    return isinstance(x, str) and x[0][0] in 'wxyz'

def function(x):
    return isinstance(x, tuple) and x[0][0] in 'abcdfghj'

def constant(x):
    return function(x) and len(x[1]) == 0
```

Predicate logic formulas are defined inductively as follows:

1. If `s` is a string beginning with 'P', 'Q', 'R' or 'S', and `[`$t_1,\ldots,t_n$`]` is a list of terms, then `(s, [`$t_1,\ldots,t_n$`])` is an atomic formula.
2. If `f` is a formula and `{`$x_1,\ldots,x_n$`}` is a set of variables then:
   a) `('forall', {`$x_1,\ldots,x_n$`}, f)` is a formula, and
   b) `('exists', {`$x_1,\ldots,x_n$`}, f)` is a formula.
3. If `f` is a formula, then `('not', f)` is a formula.
4. If `f` and `g` are formulas, then `('arrow', f, g)` is a formula.
5. If f_1, ..., f_n are formulas, then `('and', [`f_1`, ..., `f_n`])` is a formula.
6. If f_1, ..., f_n are formulas, then `('or', [`f_1`, ..., `f_n`])` is a formula.

A variable is *free* in a formula if it appears outside the scope of a quantifier. We call a formula with no free variables a *sentence*.

Two useful functions (with straightforward inductive implementations) are

```
def get_free_vars(f, bound_vars):
```

returning a set containing of all the free variabes in `f` (excluding those found in the `bound_vars` paramater) and

```
def get_funcs(f):
```

which returns a list of all function symbols (including constants) appearing in `f`.

3.2 Unification

A substitution, as we use the term here, is the replacement of a variable by a term. If σ is a substitution and t a term, we write σt to denote the result of applying σ to t.

We represent substitutions in Python by dictionaries which essentially are mappings. Dictionaries in Python are enclosed by curly braces and have comma-separated elements. Dictionary elements themselves are colon-separated key-value pairs.

A substitution may be applied to a list of terms (such as the arguments of a predicate or function). Additionally, substitutions may be composed. We use the following functions for these purposes.

```
def subst_termlist(subst, termlist):
    return [subst_term(subst, t) for t in termlist]

def subst_term(subst, term):
    if variable(term):
        return subst[term] if term in subst else term
    else:
        return (term[0], subst_termlist(subst, term[1]))

def compose_subst(s1, s2):
    comp = {}
    comp.update(s1)
    comp.update({t:subst_term(s1, s2[t]) for t in s2})
    return comp
```

A unification algorithm is an algorithm that searches for a substitution that makes two lists of terms equal. That is, given two lists $(t_1, \ldots, t_n)$ and $(u_1, \ldots, u_n)$ of terms, a unification algorithm searches for a substitution σ such that $(\sigma t_1, \ldots, \sigma t_n) = (\sigma u_1, \ldots, \sigma u_n)$. Unification will be used in the tableaux proof procedure for predicate logic that we will discuss later.

Our unification implementation is as follows. It uses the built-in `zip()` function, which works as follows. Given two lists $(a_1, \ldots, a_n)$ and $(b_1, \ldots, b_n)$, `zip()` returns essentially a list $(c_1, \ldots, c_n)$ such that $c_i = (a_i, b_i)$.

```
def unify_termlists(t1, t2):
    if len(t1) == len(t2) == 0:
        return {}
    if len(t1) != len(t2):
        return None
    else:
        subst = {}
        for (u1, u2) in zip(t1, t2):
            v1 = subst_term(subst, u1)
            v2 = subst_term(subst, u2)
            newsubst = unify_terms(v1, v2)
            if newsubst == None:
                return None
            subst = compose_subst(newsubst, subst)
        return subst

def unify_terms(t1, t2):
    if variable(t1) and variable(t2):
        if t1 == t2:
            return {}
        else:
            return {t1:t2}
    if variable(t1) and not variable(t2):
        if t1 in variables_in_term(t2):
            return None
        else:
            return {t1:t2}
    if not variable(t1) and variable(t2):
        if t2 in variables_in_term(t1):
            return None
        else:
            return {t2:t1}
    else:
        if t1[0] != t2[0]:
            return None
        else:
            return unify_termlists(t1[1], t2[1])

def variables_in_term(term):
    if variable(term):
        return {term}
    else:
        var = set()
        for t in term[1]:
            var |= variables_in_term(t)
        return var
```

3.3 Skolemization

In this section we write $\varphi(x_1, ..., x_n)$ to emphasize that φ is a formula with free variables $x_1, ..., x_n$.

A formula with free vairables is satisfiable if every sentence resulting from an assignment of those free variables is satisfiable. Thus $\varphi(x_1, ..., x_n)$ is satisfiable if and only if $\forall x_1, ..., x_n \varphi(x_1, ...x_n)$ is satisfiable. Consider the formula $\exists y \varphi(x, y)$ where x is free and y is bound by $\exists$. This formula is equivalent to $\forall x \exists y \varphi(x, y)$. Suppose that it is satisfiable in some model. In that model it must be that for every x, there is a y such that $\varphi(x, y)$ holds. In other words, φ defines a function from x to y in the model's domain. But if that is true, then there is an extension of that model where that function is given an explicit interpretation. Skolemization is the process of intruducing a new function in the formula that stands for that formula. Thus $\exists y \varphi(x, y)$ is satisfiable if and only if $\varphi(x, f(x))$ is satisfiable for some function f not appearing in φ.

More generally, if $f_1, ..., f_n$ do not appear in φ then

$$\exists y_1, ..., y_n \varphi(x_1, ..., x_m, y_1, ..., y_n) \equiv \varphi(x_1, ..., x_n, f_1(x_1, ..., x_m), ..., f_n(x_1, ..., x_m)) \text{ and}$$
$$\neg\forall y_1, ..., y_n \varphi(x_1, ..., x_m, y_1, ..., y_n) \equiv \neg\varphi(x_1, ..., x_n, f_1(x_1, ..., x_m), ..., f_n(x_1, ..., x_m)).$$

By using this Skolemization technique on formulas of one of the forms above, we reduce the cases new in the predicate satisfaction problem to formulas beginning with $\forall$ and $\neg\exists$.

The following Python takes the formula in the scope of the existential quantifier (`f`) and the variables that it ranges over (`exists_vars`), and returns the skolemized formula. The parameter `func_counter` can be used to keep track of Skolem functions that have already been used.

```python
def tableau_skolemize(f, exists_vars, func_counter):
    free_vars = get_free_vars(f, exists_vars)
    used_funcs = get_funcs(f)

    g = f
    for v in exists_vars:
        while 'f' + str(func_counter) in used_funcs:
            func_counter += 1
        skolem_func = 'f' + str(func_counter)
        used_funcs.append(skolem_func)
        g = subst_formula({v:(skolem_func, list(free_vars))}, g)
    return (g, func_counter + 1)
```

3.4 Tableaux

In the case of predicate logic, two new rules are added to handle quantifiers.

- If ψ is a delta (i.e. existential) formula, then replace it with its skolemization.
- If ψ is a gamma (i.e. universal) formula, then add to the branch a copy of ψ in which a the universally quantified variable has been replaced with a free variable that does not occur elsewhere in the tree. Thus, if $\psi = \forall x_1, \ldots, x_n \chi(x_1, \ldots, x_n)$, then add $\chi(y_1, \ldots, y_n)$ where $y_1, \ldots, y_n$ are new free variables.

In order for a branch to close, it has to contain both a formula P and $\neg P$. Obviously, this is the case if we have $P(x1, \ldots, x_n)$ and $\neg P(x1, \ldots, x_n)$. However, if we have $P(x1, \ldots, x_n)$ and $\neg P(y_1, \ldots, y_n)$, we need to apply unification to the terms of P and $\neg P$. Since we would like to close the whole tree, we need to choose unifications that not just close a single branch, but also are compatible with the unifications needed to close other branches.

In some tableau proofs, it necessary to unify a variable not only with one other term, but with two or more. For this reason, it may be necessary to apply the gamma rule to a formule more than once. Unfortunately, given the undecidability of first-order logic, often it is not known in advance how often the gamma rule should be applied. An implementation of a first-order tableaux proof procedure therefore needs to limit the number of gamma rule applications. This limit should be specified along with the formula to be proved. In our implementation, this limit is called `gdepth`.

We use the `tableau_canonize()` function to transform formulas into a canonical type. That is, alpha formulas are transformed into formulas with $\wedge$ as their main operator, gamma formulas are transformed such that $\forall$ is their main operator and so forth. To determine the type of a formula, this allows us simply to determine its main oeprator. The implementation of the function is as follows.

```python
def tableau_canonize(f):

    # Handle double negations.
    if f[0] == 'not' and f[1][0] == 'not':
        return tableau_canonize(f[1][1])

    # Handle alpha formulas.
    if f[0] == 'and':
        return f
    if f[0] == 'not' and f[1][0] == 'or':
        return ('and', [('not', g) for g in f[1][1]])
    if f[0] == 'not' and f[1][0] == 'arrow':
        return ('and', [f[1][1], ('not', f[1][2])])

    # Handle beta formulas.
    if f[0] == 'or':
        return f
    if f[0] == 'not' and f[1][0] == 'and':
        return ('or', [('not', g) for g in f[1][1]])
    if f[0] == 'arrow':
        return ('or', [('not', f[1]), f[2]])

    # Handle delta formulas.
    if f[0] == 'exists':
        return f
    if f[0] == 'not' and f[1][0] == 'all':
        return ('exists', f[1][1], ('not', f[1][2]))

    # Handle gamma formulas.
    if f[0] == 'all':
        return f
    if f[0] == 'not' and f[1][0] == 'exists':
        return ('all', f[1][1], ('not', f[1][2]))

    # Handle literals.
    else:
        return f
```

The beginning of our implementation, then, is as follows.

```
def tableau(f, gdepth):
    branches = tableau_expand(f, gdepth)
    return tableau_closed(branches)

def tableau_expand(f, qdepth):
    branch = [tableau_canonize(('not', f))]
    return tableau_expand_do([branch], qdepth, [], 0, 0)
```

The real work is done by `tableau_expand_do()`. This function takes the full tableau as a list of branches, the 'gdepth', a counter for the Skolem functions and a counter for the free variables. These two counters are used to ensure newly introduced Skolem functions and free variables are unique.

```
def tableau_expand_do(branches, gdepth, gcounter, skolem_func_counter, \
uni_var_counter):
    for branch in branches[:]:
        # Handle alpha formulas.
        for f in branch:

            if f[0] == 'and':
                branch.remove(f)
                subs = [tableau_canonize(g) for g in f[1]]
                branch += subs
                return tableau_expand_do(branches, gdepth, gcounter, \
                    skolem_func_counter, uni_var_counter)
        # Handle beta formulas.
        for f in branch:

            if f[0] == 'or':
                branches.remove(branch)
                branch.remove(f)
                subs = [tableau_canonize(g) for g in f[1]]
                for g in subs:
                    branches.append(branch + [g])
                return tableau_expand_do(branches, gdepth, gcounter, \
                    skolem_func_counter, uni_var_counter)

        # Handle delta formulas (before gamma formulas).
        for f in branch:

            if f[0] == 'exists':
                branch.remove(f)
                g, skolem_func_counter = tableau_skolemize(f[2], f[1],
                    skolem_func_counter)
                branch.append(tableau_canonize(g))
                return tableau_expand_do(branches, gdepth, gcounter, \
                    skolem_func_counter, uni_var_counter)

        # Handle gamma formulas.
        for f in branch:

            if f[0] == 'all':
                gcounter.append(f)
                branch.remove(f)
                if gcounter.count(f) < gdepth:
                    branch.append(f)
                new_f = f[2]
                for var in f[1]:
                    parameter =  'x' + str(uni_var_counter)
                    uni_var_counter += 1
                    new_f = subst_formula({var:parameter}, new_f)
                branch.append(tableau_canonize(new_f))
                return tableau_expand_do(branches, gdepth, gcounter, \
                    skolem_func_counter, uni_var_counter)

    return branches
```

This function expands all branches in the tableau by repeated application of the expansion rules. Gamma rules are applied until a 'gdepth' of 0 has been reached. If the tableau expansion is complete, the algorithm moves to the closure stage. In this stage we visit all branches of the tree, and try to close each of them, taking into account all substitutions from previous branch closures.

```
def tableau_closed(branches):
    substs = {}
    for branch in branches:
        newsubsts = tableau_branch_closed(branch, substs)
        if newsubsts == None:
            return False
        substs.update(newsubsts)
    return True

def tableau_branch_closed(branch, substs):
    positives = [f for f in branch if pred(f)]
    negatives = [f for f in branch if f[0] == 'not']

    for f in positives:
        for g in negatives:
            if f[0] == g[1][0]:
                f_terms = subst_termlist(substs, f[1])
                g_terms = subst_termlist(substs, g[1][1])
                newsubsts = unify_termlists(f_terms, g_terms)
                if newsubsts != None:
                    return newsubsts
    return None
```