assignment2

November 23, 2017

```
In [ ]: # -*- coding: utf-8 -*-
        import re
        from collections import Counter
        from copy import copy


        class Literal(object):
            def __init__(self, name):
                """
                Literal class representing both positive and negative literals.

                @param name: Name of literal
                @ivar sign: True if literal is positive, False if literal is negative
                """
                if name[0] in ('-', '', '~', '!', 'ň'):
                    self.name = name[1:]
                    self.sign = False
                else:
                    self.name = name
                    self.sign = True

            def __str__(self):
                return self.name if self.sign else '{}'.format(self.name)

            def __repr__(self):
                return self.name

            def __neg__(self):
                negation = '-' if self.sign else ''
                return Literal('{}{}'.format(negation, self.name))

            def __eq__(self, other):
                if self.__class__ != other.__class__:
                    return False
                else:
                    # Two symbols with the same name are the same symbol,
                    # regardless of whether they're positive or negative
                    return self.name == other.name

            def __lt__(self, other):
                if self.__class__ != other.__class__:
                    return False
                else:
                    return self.name < other.name

            def __hash__(self):
                return hash(self.name)


        # The next ~200 lines are a CNF parser.
        # I wrote it so I could read in knowledge bases from text files in the exact
        # format I want, and to practice a bit with regex, but feel free to ignore
        # if you're short on time.
        # Ideally I'd split this out into another file, but since I can only
        # upload 2 files onto the platform, it stays all in this one file... sorry.
        class InvalidLiteralNameError(ValueError):
            def __init__(self, sentence, unrecognized_symbol, position):
                self.message = 'Knowledge base sentence "{}" contains ' \
                               'non-alphanumeric literal name "{}" at position {}' \
                               .format(sentence, unrecognized_symbol, position)
                super().__init__(self.message)


        class InvalidSentenceFormatError(ValueError):
            def __init__(self, sentence, literal1, literal2):
                self.message = 'Knowledge base sentence "{}" has no logical ' \
                               'connector separating "{}" and "{}". Available ' \
                               'logical connectors: , <->, <=>, , =>, ->, , ' \
                               'v, ||, , ^, &&' \
                               .format(sentence, literal1, literal2)
                super().__init__(self.message)


        def eliminate_implications(sentence):
            """
            Eliminates implications in a given logical sentence by substituting in
            logical equivalences for biconditionals and implications, i.e.:
                >>> print(eliminate_implications('A  B'))
                '(A  B)  (B  A)'
                >>> print(eliminate_implications('C  D'))
                'C  D'

            @param sentence: Valid logical sentence (see validate_sentence() for rules
                             on validity)
            @type sentence: str
            @return: Logical sentence with implications eliminated, as in above example
            @rtype: str
            """
            sentence_LHS = re.sub(
                r'(.*\S)\s*\s*(.*\S)',
                r'\1  \2',
                re.sub(
                    r'(.*\S)\s*\s*(.*\S)',
                    r'\1  \2',
                    sentence
                )
            )

            if '' in sentence:
                sentence_RHS = re.sub(
                    r'(.*\S)\s*\s*(.*\S)',
                    r'\1  \2',
                    re.sub(
                        r'(.*\S)\s*\s*(.*\S)',
                        r'\2  \1',
                        sentence
                    )
                )
                sentence = '({})  ({})'.format(sentence_LHS, sentence_RHS)
            else:
                sentence = sentence_LHS

            return sentence


        def move_negation_in(sentence):
            """
            Moves negation inwards (i.e., directly next to the relevant literal/s) by
            applying double-negation elimination and De Morgan's rules:
                >>> print(move_negation_in('(A)'))
                'A'
                >>> print(move_negation_in('(B  C)'))
                'B  C'
                >>> print(move_negation_in('(D  E)'))
                'D  E'

            @param sentence: Valid logical sentence, with implications eliminated
                             (see validate_sentence() for rules on validity)
            @type sentence: str

            @return: Logical sentence with negation moved in, as in above examples
            @rtype: str
            """
            sentence = re.sub(r'\(*(\w)\)*', r'\1', sentence)
            sentence = re.sub(r'\((\w)\s*\s*(\w)\)', r'(\1  \2)', sentence)
            sentence = re.sub(r'\((\w)\s*\s*(\w)\)', r'(\1  \2)', sentence)

            return sentence


        def distribute_or_over_and(sentence):
            """
            Distributes or over and according to the logical equivalence, i.e.:
                >>> print(distribute_or_over_and('A   (B   C)'))
                '(B   A)   (C   A)'
                >>> print(distribute_or_over_and('(B   C)   A'))
                '(B   A)   (C   A)'

            @param sentence: Valid logical sentence, with implications eliminated
                             and negation moved in (see validate_sentence() for
                             rules on validity)
            @type sentence: str

            @return: Logical sentence with  distributed over , as in above example
            @rtype: str
            """
            sentence = re.sub(
                r'(*\w)\s*\s*\((*\w)\s*\s*(*\w)\)',
                r'(\2  \1)  (\3  \1)',
                sentence
            )
            sentence = re.sub(
                r'\((*\w)\s*\s*(*\w)\)\s*\s*(*\w)',
                r'(\1  \3)  (\2  \3)',
                sentence
            )

            return sentence


        def validate_sentence(sentence):
            """
            Validates given knowledge base sentence. Gracefully fixes poorly grouped
            literals ăe.g. C   F   B becomes (C   F)   B  but otherwise assumes
            brackets are correctly matched.

            @param sentence: Knowledge base sentence to be validated
            @type sentence: str

            @raise InvalidLiteralNameError: Raised if sentence contains
```

```
                                non-alphanumeric literal names (including
                                underscores and unicode letters, but no
                                spaces)
@raise InvalidSentenceFormatError: Raised if sentence doesn't have any of
                                   , , , or  separating each literal.

@return: Valid knowledge base sentence
@rtype: str
"""
split_sentence = list(filter(None, re.split('[ ()]', sentence)))

if not (sentence[0].isalnum() or sentence[1].isalnum()):
    raise InvalidLiteralNameError(sentence, sentence[0], 0)
else:
    previous_symbol = split_sentence[0]

for i in range(1, len(split_sentence)):
    symbol = split_sentence[i]
    if previous_symbol in ('', '', '', ''):
        if not symbol.replace('_', '').isalnum():
            raise InvalidLiteralNameError(sentence, previous_symbol, i-1)

    elif previous_symbol.replace('_', '').isalnum():
        if symbol not in ('', '', '', ''):
            raise InvalidSentenceFormatError(
                sentence, previous_symbol, symbol
            )

    else:
        raise InvalidLiteralNameError(sentence, previous_symbol, i-1)

    previous_symbol = symbol

# TODO: convert this to regex
if '' in sentence:
    implication_idx = sentence.index('')
    left = sentence[:implication_idx].strip()
    right = sentence[implication_idx+1:].strip()

    if len(left) > 2 and left[0] != '(':
        left = '({})'.format(left)

    if len(right) > 2 and right[0] != '(':
        right = '({})'.format(right)

    sentence = '{}  {}'.format(left, right)

if '' in sentence:
```

```
            implication_idx = sentence.index('')
            left = sentence[:implication_idx].strip()
            right = sentence[implication_idx+1:].strip()

            if len(left) > 2 and left[0] != '(':
                left = '({})'.format(left)

            if len(right) > 2 and right[0] != '(':
                right = '({})'.format(right)

            sentence = '{}  {}'.format(left, right)

        return sentence


    def format_sentence(sentence):
        """
        Formats sentence into the correct Python representation/types for our
        use, i.e. elements of disjunctions become Literals, each disjunction is a
        set, and each element of a conjunction is a set inside the overall list.

        @param sentence: Valid logical sentence, with implications eliminated,
                         negation moved in, and ors distributed over ands
                         (see validate_sentence() for rules on validity)
        @type sentence: str

        @return: Python object representation of sentence
        @rtype: list[set(Literal)]
        """
        disjunctions = sentence.split('')
        conjunction = []

        for disjunction in disjunctions:
            disjunction = disjunction.strip().replace('(', '').replace(')', '')

            if '' in disjunction:
                literals = disjunction.split('')
                formatted_disjunction = {Literal(literal.strip())
                                         for literal in literals}
            else:
                formatted_disjunction = {Literal(disjunction)}

            conjunction.append(formatted_disjunction)

        return conjunction


    def parse_CNF(knowledge_base):
```

```
"""
String parser which extracts knowledge base as list of sets of Literals
in conjunctive normal form.

@param knowledge_base: Logical sentences separated by newlines,
                       using /<=>/<-> for biconditionals,
                       /=>/==>/-> for implication, /v/||/| for logical
                       disjunction, /^/&&/& for logical
                       conjunction, and /ň/~/!/- for logical negation.
@type knowledge_base: str

@return: knowledge_base in conjunctive normal form, with Literals in sets
         representing disjunction, and sets in a list representing
         conjunction
@rtype: list[set(Literal)]
"""
# Clean up given KB so we only have to look out for one set of logical
# connectors
knowledge_base = re.sub(
    r'ň|~|!|-', r'',
    re.sub(
        r'\^|&+', r'',
        re.sub(
            r'v|\|+', r'',
            re.sub(
                r'=>|->|==>', r'',
                re.sub(
                    r'<=>|<->', r'',
                    knowledge_base
                )
            )
        )
    )
)

sentences = knowledge_base.split('\n')
formatted_knowledge_base = []

for sentence in sentences:
    sentence = sentence.strip()
    sentence = validate_sentence(sentence)
    sentence = eliminate_implications(sentence)
    sentence = move_negation_in(sentence)
    sentence = distribute_or_over_and(sentence)
    sentence = format_sentence(sentence)

    formatted_knowledge_base.extend(sentence)
```

```python
    return formatted_knowledge_base


def clause_true(clause, model):
    """
    Checks whether given clause is True according to symbol/value assignments
    in given model. Since clause is a disjunction of Literals, only one
    Literal in clause has to be True for the entire clause to be True.

    @param clause: One knowledge base clause, a disjunction of Literals
    @type clause: set(Literal)
    @param model: Model currently being tested to see whether it makes clause
                  True. All keys are names of Literals and all values
                  are either 'true' or 'false', depending on whether the
                  literal is currently True or False in the model we're
                  testing.

    @return: True if clause is True according to model, otherwise False
    @rtype: bool
    """
    for symbol in clause:
        if (
            symbol in model.keys() and
            model[symbol] == str(symbol.sign).lower()
        ):
            return True

    return False


def degree_heuristic_sort(**kwargs):
    """
    Order knowledge base symbols in order of frequency. Symbols with the same
    frequency are ordered arbitrarily.

    @param unknown_clauses: List of sets of Literals representing clauses which
                            aren't yet True according to a tested model, where
                            sets are disjunctions, and the overall list is a
                            conjunction.
    @type unknown_clauses: list[set(Literal)]
    @param unused_symbols: List of symbols present in knowledge_base which
                           haven't yet been assigned either 'true' or 'false'
                           in model.

    @return unused_symbols: Symbols appearing in unknown_clauses in order of
                            their frequency of appearance.
    @rtype: list[Literal]
    """
    unknown_clauses = kwargs['unknown_clauses']
    unused_symbols = kwargs['unused_symbols']

    flat_clauses = [symbol for clause in unknown_clauses for symbol in clause]
    frequencies = Counter(flat_clauses)

    unused_symbols = sorted(
        unused_symbols,
        # symbols eventually assigned 'free' won't appear in frequencies
        key=lambda symbol: frequencies.get(symbol, 0),
        reverse=True
    )

    return unused_symbols


def get_pure_symbol(**kwargs):
    """
    Gets the pure symbol that hasn't yet been assigned in the model and
    which occurs the most over all remaining unknown clauses.
    Pure symbols are those which appear either as only the positive or as only
    the negative literal. Ignores clauses which have already been proven True.

    @param unknown_clauses: List of sets of Literals representing clauses which
                            aren't yet True according to a tested model, where
                            sets are disjunctions, and the overall list is a
                            conjunction.
    @param unused_symbols: Symbols appearing in unknown_clauses.
                           If DPLL_Satisfiable is passed degree_heuristic=True
                           then these symbols are in order of their frequency
                           of appearance in unknown_clauses, otherwise,
                           these symbols are ordered arbitrarily.

    @return: The first unused symbol which appears as only the positive or
             only the negative literal in unknown_clauses, if it exists; and
             the value that this symbol should be set to in the model so as to
             make at least one unknown clause True ('true' if symbol only
             appears as the positive literal; 'false' if the symbol only
             appears as the negative literal).
             If this symbol doesn't exist, then None for both the unused symbol
             and its value.
    """
    unknown_clauses = kwargs['unknown_clauses']
    unused_symbols = degree_heuristic_sort(
        unknown_clauses=unknown_clauses,
        unused_symbols=kwargs['unused_symbols']
    )
    for symbol in unused_symbols:
        # I briefly thought about doing this without these extra two variables,
        # but decided my alternative involved trading off a small amount of
        # memory usage for an increased scaling constant and wound up being
        # less readable, so I went for this version. (I was thinking of having
        # one for comprehrension to create a list of the signs of the symbol
        # in each clause, but then realized I'd have to iterate over my list
        # again with all() anyway to determine symbol purity.)
        appears_pos = False
        appears_neg = False

        for clause in unknown_clauses:
            if symbol in clause:
                symbol_in_clause = list(clause)[list(clause).index(symbol)]
                if not appears_pos and symbol_in_clause.sign:
                    appears_pos = True
                elif not appears_neg and not symbol_in_clause.sign:
                    appears_neg = True

        if appears_pos ^ appears_neg:
            return symbol, str(appears_pos).lower()

    return None, None


def get_unit_clause(**kwargs):
    """
    Gets the first unit clause from the remaining unproven clauses.
    A unit clause is one in which all literals except one have already been
    assigned false (i.e. there is only one non-false literal, and it hasn't
    yet been assigned).

    @param unknown_clauses: List of sets of Literals representing clauses which
                            aren't yet True according to a tested model, where
                            sets are disjunctions, and the overall list is a
                            conjunction.
    @param model: Model currently being tested to see whether it satisfies the
                  knowledge base given to DPLL(). All keys are names of
                  Literals and all values are either 'true' or 'false',
                  depending on whether the literal is currently true or false
                  in the model we're checking for satisfiability.

    @return: The only symbol in the first unit clause in the remaining
             unproven clauses, and the value that must be assigned to this
             symbol in the model to make this unit clause True, provided such a
             symbol exists.
             If this symbol doesn't exist, then None for both the symbol and
             its value.
    """
    unknown_clauses = kwargs['unknown_clauses']
    model = kwargs['model']

    for clause in unknown_clauses:
        unassigned_symbols = [
            symbol for symbol in clause
            if symbol not in model.keys()
        ]
        if len(unassigned_symbols) == 1:
            unused_symbol = unassigned_symbols[0]
            return unused_symbol, str(unused_symbol.sign).lower()

    return None, None


def DPLL(knowledge_base, unused_symbols, model, heuristics):
    """
    Davis-Putnam-Logemann-Loveland algorithm for checking satisfiability of
    a knowledge base given in propositional logic.

    @param knowledge_base: List of sets of Literals. Every Literal in a set is
                           interpreted as being in disjunction with every other
                           Literal in the set; every set is interpreted as
                           being in conjunction with every other set.
    @param unused_symbols: List of symbols present in knowledge_base which
                           haven't yet been assigned either 'true' or 'false'
                           in model.
    @param model: Model currently being tested to see whether it satisfies
                  knowledge_base. All keys are names of Literals and all values
                  are either 'true' or 'false', depending on whether the
                  literal is currently true or false in the model we're
                  checking for satisfiability.
    @param heuristics: List of function handlers of heuristics to apply when
                       choosing symbols for assignment in model.

    @return satisfiable: True if knowledge_base is satisfiable, else False.
    @return model: If satisfiable, the first model which satisfies
                   knowledge_base, where all keys are names of Literals and all
                   values are either 'true', 'false', or 'free', depending on
                   whether the literal must be true, false, or can be freely
                   chosen to ensure satisfiability.
                   If not satisfiable, empty dictionary {}.
    """
    unknown_clauses = []

    for clause in knowledge_base:
        if not clause_true(clause, model):
```

```python
        unknown_clauses.append(clause)

if not unknown_clauses:
    # Any remaining unused symbols are assigned as 'free'
    # since regardless of their truth value, the knowledge base
    # is still satisfiable.
    model.update({symbol: 'free' for symbol in unused_symbols})
    return True, model
elif not unused_symbols:
    return False, model

for heuristic in heuristics:
    if heuristic != degree_heuristic_sort:
        unused_symbol, value = heuristic(
            unknown_clauses=unknown_clauses,
            unused_symbols=unused_symbols,
            model=model
        )

        if unused_symbol:
            model_with_unused = copy(model)
            model_with_unused[unused_symbol] = value
            rest = list(filter(
                lambda symbol: symbol != unused_symbol, unused_symbols
            ))

            return DPLL(
                knowledge_base,
                rest,
                model_with_unused,
                heuristics
            )
    else:
        unused_symbols = degree_heuristic_sort(
            unknown_clauses=unknown_clauses,
            unused_symbols=unused_symbols
        )

# At least using Python 3 gets us this new iterable unpacking hotness
unused_symbol, *rest = unused_symbols if unused_symbols else [None]

model_true_unused = copy(model)
model_true_unused[unused_symbol] = 'true'
model_false_unused = copy(model)
model_false_unused[unused_symbol] = 'false'

satisfiable_model_true, model_true = DPLL(
    knowledge_base,
            rest,
            model_true_unused,
            heuristics
        )
        satisfiable_model_false, model_false = DPLL(
            knowledge_base,
            rest,
            model_false_unused,
            heuristics
        )

        if satisfiable_model_true:
            return satisfiable_model_true, model_true
        elif satisfiable_model_false:
            return satisfiable_model_false, model_false
        else:
            return False, {}


    def DPLL_Satisfiable(knowledge_base, heuristics=[]):
        """
        User-friendly wrapper for DPLL which allows us to pass in only the
        knowledge base, rather than having to pass in the knowledge base,
        the symbols present, and an empty model. This wrapper also lets us parse
        string knowledge bases into conjunctive normal form, validate a given
        knowledge base, and use various symbol assigning heuristics.

        @param knowledge_base: Either:
                               - list of list of sets of Literals. Every Literal
                                 in a set is interpreted as being in disjunction
                                 with every other Literal in the set; every set is
                                 interpreted as being in conjunction with every
                                 other set; or
                               - Logical sentences using /<=>/<-> for
                                 biconditionals, /=>/==>/-> for implication,
                                 /v/|| for logical disjunction, /ˆ/&& for logical
                                 conjunction, and /ň/~/!/- for logical negation.
        @type knowledge_base: list[list[set(Literal)]] or str
        @param heuristics: List of function handlers of heuristics to apply when
                           choosing symbols for assignment in model. (default: [])

        @raise TypeError: Raised if knowledge_base is not of type str or an
                          instance of type list.

        @return satisfiable: True if knowledge_base is satisfiable, else False.
        @return model: If satisfiable, the first model which satisfies
                       knowledge_base, where all keys are Literals and all values
                       are either 'true', 'false', or 'free', depending on whether
```

```
                    the literal must be true, false, or can be freely chosen
                    to ensure satisfiability.
                    If not satisfiable, empty dictionary {}.
    """
    if type(knowledge_base) == str:
        knowledge_base = parse_CNF(knowledge_base)

    if isinstance(knowledge_base, list):  # so subclasses of list work too
        unused_symbols = list(set(
            [symbol for clause in knowledge_base for symbol in clause]
        ))

    else:
        raise TypeError('Knowledge base {} is not of type str, or an '
                        'instance of type list.'.format(knowledge_base))

    return DPLL(knowledge_base, unused_symbols, {}, heuristics)


def test_DPLL_Satisfiable(knowledge_bases):
    """
    Tests DPLL_Satisfiable, printing out whether each given knowledge base is
    satisfiable or not, as well as the first model which satisfies the
    knowledge base (if it is indeed satisfiable).

    @param knowledge_bases: List of strings, or list of lists, representing
                            distinct knowledge bases. No relation is assumed
                            between each set of lists or strings.
    @type knowledge_bases:  list[str or list[set(Literal)]]
    """
    for knowledge_base in knowledge_bases:
        satisfiable, model = DPLL_Satisfiable(
            knowledge_base,
            heuristics=[
                get_pure_symbol,
                degree_heuristic_sort,
                get_unit_clause
            ]
        )
        print('\nKnowledge base {} is{} satisfiable.{}'.format(
            knowledge_base,
            '' if satisfiable else ' not',
            '\nThe first model satisfying the knowledge base is {}'.format(
                model
            ) if satisfiable else ''
        ))
```

```python
A = Literal('A')
B = Literal('B')
C = Literal('C')
D = Literal('D')
KB = [{A, B}, {A, -C}, {-A, B, D}]  # example KB from assignment instructions

test_DPLL_Satisfiable([
    # KB from exercise 7.20 in Russell & Norvig (2009)
    'A <=> (B | E)\n \
     E ==> D\n \
     C & F ==> ~B\n \
     E ==> B\n \
     B ==> F\n \
     B ==> C',
    KB
])
```