CS152 - DPLL Algorithm

Anna Pauxberger

04 April 2019

1. Introduction

1.1. The DPLL Implementation

The following shows the code implementation and results of the DPLL algorithm. It finds whether a knowledge base is satisfiable, and if it is returns the model for which it is satisfiable. It implements heuristics (unit clause, pure symbol, degree heuristic) and shows that the use of such heuristics improves the performance of the algorithm.

Code and analysis can be found here as well: `https://github.com/annapaux/DPLL-algorithm`

1.2. HCs

I would like to draw your attention to the following Habits of Mind and Foundational Concepts:

- **#algorithms**: I effectively implement a complex algorithm in Python, extend it with heuristics and explain its functioning with structured comments at every step.
- **#deduction**: Deductive reasoning is deterministic, and occurs whenever a conclusion logically follows from a set of axioms. The unit clause heuristic, for example, is a propositional resolution relying in deductive reasoning: once we have a unit clause, and we place a sign such that it resolves to True, this can initiate a cascade of assignments.
- **#breakitdown**: I broke down the problem into tractable components (DPLLSatisfiable() initiates the search, DPLL() iteratively searches the solution, the heuristic functions optimize) and document them in a clear and structured manner.
- **#optimize**: I improve the DPLL algorithm by implementing heuristics (unit clause, pure symbol, degree heuristics) that make the algorithm more efficient by choosing symbols that more quickly lead to a solution. As you can see in the output, the heuristics improve the time to run the algorithm from an average 0.008 to 0.004 seconds. (I think the intended understanding of this HCs refers to parameter optimization, though I think the abstracted meaning of optimization in terms of improving results can be applied here.)
- **#professionalism**: I present the work professionally, adhering to academic guidelines and in a well structured and easily readable document.
- **#audience**: I tailored the submission towards' your specified guidelines with regards to submission details of the algorithm and paper submission.

2. Code Implementation

2.1. Preparing for DPLL

```
import random
import copy
import time
import numpy as np


class Literal():
    '''
    Represents a logicial literal, with name and sign. Example: Literal("A")
    '''

    def __init__(self, literal, sign = True):
        '''
        Initialize the literal with positive sign, unless indicated
        otherwise via the __neg__ operator.
        '''
        self.name = str(literal)
        self.sign = sign

    def __repr__(self):
        '''Prints the name of the literal including a '-' for neg. literals. '''
        return self.name if self.sign else "-" + self.name

    def __neg__(self):
        '''When a dash/ minus is in front of the Literal, flip the sign.'''
        return Literal(self.name, sign = not self.sign)

    def __hash__(self):
        '''When hashed (e.g. in the dictionary or set), the name is hashed.'''
        return hash(self.name)

    def __eq__(self, other):
        '''
        When equality operator is applied (e.g. if Literal is in a set)
        compare the name.
        '''
        return self.name == other.name
```

2.2. DPLL Algorithm

```
########################################
######### DPLL Algorithm ###############
########################################

def DPLLSatisfiable(KB, use_print=False,
                    use_unit_clause=True,
                    use_pure_symbol=True,
                    use_degree_heuristic=True):
    '''
    Defines clauses, symbols and an empty model from the KB. Returns whether
    or not the KB is satisfiable and if yes one model that satisfies the KB.
    The KB is a conjunction of disjunctions of atoms.
```

```
    Example KB input: {A, B}, {A, -C}, {-A, B, D}]
    Example satisfiable output: True
    Example model output: {A: true, B: true, C: false, D: free}
    '''

    # define set of clauses and symbols from KB
    clauses = KB
    symbols = set()
    for clause in KB:
        for literal in clause:
            symbols.add(literal)
    symbols = list(symbols)

    # define an empty model
    model = {}

    # print start of model
    if use_print:
        print('---- START DPLL ----')
        print(f'knowledge base clauses: {KB}')
        print(f'symbols: {symbols}')
        print('Now recursing ...')

    # check satisfiability and explore models with DPLL()
    satisfiable, model = DPLL(clauses, symbols, model, use_print=use_print,
                              use_unit_clause=use_unit_clause,
                              use_pure_symbol=use_pure_symbol,
                              use_degree_heuristic=use_degree_heuristic)

    # any literal that is not assigned a value is free (either true or false)
    # the KB is satisfiable regardless of the assignment of the symbol
    for s in symbols:
        if s not in model:
            model[s] = 'free'
    if use_print:
        print('------ RESULT ------')
        print('satisfiable:', satisfiable)
        print('model:', model)
        print('')

    return satisfiable, model


def DPLL(clauses, symbols, model, use_print=False,
        use_unit_clause=True,
        use_pure_symbol=True,
        use_degree_heuristic=True):
    '''
    Recursive model that checks satisfiability of a model.

    It first checks whether, given the current model, the KB is
    satisfiable, meaning that all clauses evaluate to True. If that is the case,
    it returns satisfiable=True and model=model for which the KB is satisfiable.
    If it is not satisfiable, it tries different combinations of symbol assign-
    ments recursively:
```

```
1) Pick a symbol
2) Assign 'true'
3) Check if clauses are satisfiable
4) a) if all clauses are True, return the (satisfiable, model)
   b) if any clause is False, recurse the tree up and try the same
      symbol with assignment 'false', if that fails too, recurse to the
      previous symbol
   c) if clauses are undetermined, continue recursion with the next symbol

As soon as one symbol in the clause is true, the clause is true, since the
clauses are conjunctions of disjunctions. For (A v B) to be true, either A
or B can be true.
However, we need all clauses to be true for the KB to be satisfied. For
[(A V B)  (C V D)] to be true, (A V B) has to be true and (C V D) has to
be true.
'''
# CHECK SATISFIABILITY
# determined if all clauses are satisfied ('true')
# list of clause status: 'true', 'false', 'undetermined'
clause_status_list = []
for clause in clauses:
    clause_status = None
    for literal in clause:
        if literal in model:
            # if sign and truth value are true, the clause is true
            if literal.sign and (model[literal] == 'true'):
                clause_status = 'true'
            # if sign and truth value are false, the clause is true
            elif not literal.sign and (model[literal] == 'false'):
                    clause_status = 'true'
            # if not, and the clause_status is None (not true or false),
            # the clause is false
            else:
                if clause_status is None:
                  clause_status = 'false'
        # if literal is not in the model and the clause not true,
        # it is undetermined
        else:
            if clause_status != 'true':
              clause_status = 'undetermined'
    clause_status_list.append(clause_status)

if use_print:
    print('- clause statuses:', clause_status_list)
    print('- model:', model)
    print('- symbols left:', symbols)
    print('')


# END RECURSION (BASE CASE)

# if all clauses are true, the KB is satisfiable
kb_is_true = all(c == 'true' for c in clause_status_list)
# if any clause is false, the KB is not satisfiable
```

```
kb_is_false = any(c == 'false' for c in clause_status_list)

if kb_is_true:
    return True, model
elif kb_is_false:
    return False, model


# CONTINUE RECURSION
# else, if clauses are undetermined, we continue recursion
else:
    # find the next symbol and start by setting it to None
    next_symbol = None
    # to be more efficient, we only explore the clauses that are undefined
    undefined_clauses = []
    for i in range(len(clause_status_list)):
        if clause_status_list[i] == 'undetermined':
            undefined_clauses.append(clauses[i])

    # implement heuristics to choose the next symbol
    # default arguments specify what heuristics to consider
    # default: first try to find a unit clause, then a pure symbol, then the
    # onewith highest clause occurrence or randomly.

    # (1) FIND UNIT CLAUSES (UNIT CLAUSE HEURISTIC)
    if use_unit_clause and next_symbol is None:
        next_symbol, remaining_symbols = unit_clause(undefined_clauses,
                                                     symbols, model)
        if use_print and next_symbol:
            print('Unit clause heuristic chooses next literal:', next_symbol)

    # (2) DETERMINE PURE SYMBOLS (PURE SYMBOL HEURISTIC)
    if use_pure_symbol and next_symbol is None:
        next_symbol, remaining_symbols = pure_symbol(undefined_clauses,
                                                     symbols)
        if use_print and next_symbol:
            print('Pure symbol heuristic chooses next literal:', next_symbol)

    # (3) CHOOSE SYMBOL IN MOST SENTENCES (DEGREE HEURISTIC)
    if use_degree_heuristic and next_symbol is None:
        next_symbol, remaining_symbols = degree_heuristic(
                                            undefined_clauses, symbols)
        if use_print and next_symbol:
            print('Degree heuristic chooses next symbol:', next_symbol)

    # (4) CHOOSE RANDOM SYMBOL
    elif next_symbol is None:
        next_symbol, remaining_symbols = random_symbol(symbols)
        if use_print and next_symbol:
            print('A symbol is randomly chosen:', net_symbol)

    # add the next symbol with first 'true' and then 'false' assignment
    # to the mdoel
    model[next_symbol] = 'true'
```

```
        # create a deep copy of the clauses, symbols and model to ensure that
        # the recursion refers to different object instances at each recursive call
        satisfiable, next_model = DPLL(copy.deepcopy(clauses),
                                       copy.deepcopy(symbols),
                                       copy.deepcopy(model),
                                       use_print=use_print,
                                       use_unit_clause=use_unit_clause,
                                       use_pure_symbol=use_pure_symbol,
                                       use_degree_heuristic=use_degree_heuristic)

        # if it is satisfiable with the assignment (true), return the result
        if satisfiable:
            return satisfiable, next_model

        # if it is not satisfiable, try the opposite truth assignment (false)
        elif not satisfiable:
            model[next_symbol] = 'false'
            return DPLL(copy.deepcopy(clauses),
                        copy.deepcopy(symbols),
                        copy.deepcopy(model),
                        use_print=use_print,
                        use_unit_clause=use_unit_clause,
                        use_pure_symbol=use_pure_symbol,
                        use_degree_heuristic=use_degree_heuristic)
```

2.3. Heuristic Functions

```
########################################
########## HEURISTICS ################
########################################


def unit_clause(clauses, symbols, model):
    '''
    Find unit_clauses in clauses and return its symbol as next symbol.
    Unit clauses exist when a single symbol remains unassigned.
    '''

    for clause in clauses:
        symbol_not_assigned = []
        for symbol in clause:
            if symbol in model:
                symbol_not_assigned.append(False)
            else:
                symbol_not_assigned.append(True)

                # assign the symbol as next_symbol
                next_symbol = symbol

        # and return it if it is the only unassigned symbol (sum == 1)
        if sum(symbol_not_assigned) == 1:
            symbols.remove(next_symbol)
            return next_symbol, symbols

    # if no unit clause exists return None
```

```
    return None, None


def pure_symbol(clauses, symbols):
    '''
    Find a pure symbol in the clauses and return it if it exists.
    Pure symbols occurr with one sign only. (Either all pos. or all neg.)
    '''

    next_symbol = None
    symbol_sign_dict = {}  # key=symbol, value=[list of signs]
    symbols_set = set(symbols)  # use set of symbols for efficient look-up

    for clause in clauses:
        for symbol in clause:
            # if the symbol is still in the symbols set
            if symbol in symbols_set:
                # and already occurred, thus is in the dictionary
                if symbol in symbol_sign_dict:
                    # add the symbol + sign to the dictionary
                    symbol_sign_dict[symbol].append(symbol.sign)
                else:
                    # create dictionary entry with symbol and its sign
                    symbol_sign_dict[symbol] = [symbol.sign]

    # check if there is a symbol for which all sign entries are identical
    for symbol, symbol_signs in symbol_sign_dict.items():
        if (all(sign == True for sign in symbol_signs) or
            all(sign == False for sign in symbol_signs)):
            next_symbol = symbol
            symbols.remove(next_symbol)
            return next_symbol, symbols

        # if no pure symbol exists return None, None
        else:
            return None, None


def random_symbol(symbols):
    '''Returns randomly chosen symbol and list of remaining symbols.'''
    next_symbol = random.choice(symbols)
    symbols.remove(next_symbol)
    return next_symbol, symbols


def degree_heuristic(clauses, symbols):
    '''Returns the symbol appearing in most clauses.'''

    # count the number of clauses a symbol appears in
    symbol_count = []
    for symbol in symbols:
        count = 0
        for clause in clauses:
            if symbol in clause:
                count += 1
```

```
        symbol_count.append(count)

    # choose the symbol that occurs in most sentences
    max_index = symbol_count.index(max(symbol_count))
    next_symbol = symbols[max_index]
    symbols.remove(next_symbol)
    return next_symbol, symbols
```

2.4. Running and Comparing Models

```
########################################
######## COMPARING MODELS #############
########################################

def compare_models(KB):
    print('Comparing models ...')
    without_heuristics, with_heuristics = [], []
    for i in range(100):
        start = time.time()
        DPLLSatisfiable(KB, use_unit_clause=False, use_pure_symbol=False,
                            use_degree_heuristic=False)
        end = time.time()
        without_heuristics.append(end - start)

        start = time.time()
        DPLLSatisfiable(KB, use_unit_clause=True, use_pure_symbol=True,
                            use_degree_heuristic=True)
        end = time.time()
        with_heuristics.append(end - start)

    print('Without heuristics: mean = {0:.5f}, std = {1:.5f}'.format(
        np.mean(without_heuristics), np.std(without_heuristics)))
    print('With heuristics: mean = {0:.5f}, std = {1:.5f}'.format(
        np.mean(with_heuristics), np.std(with_heuristics)))


def main():
    ########################################
    ########### TEST CASES ################
    ########################################

    A = Literal("A")
    B = Literal("B")
    C = Literal("C")
    D = Literal("D")
    E = Literal("E")
    F = Literal("F")

    ########### TEST CASE 1 ################
    KB = [{A, B}, {A, -C}, {-A, B, D}]
    satisfiable, model = DPLLSatisfiable(KB, use_print=True)
    print('')

    ########### TEST CASE 2 ################
    KB = [{-A, B, E}, {A,-B}, {A,-E}, {-E,D}, {-C, -F, -B}, {-E, B}, {-B, F},
```

```
        {-B, C}]
    satisfiable, model = DPLLSatisfiable(KB, use_print=True)
    print('')

    ########## COMPARE MODELS ##############
    KB = [{-A, B, E}, {A,-B}, {A,-E}, {-E,D}, {-C, -F, -B}, {-E, B}, {-B, F},
        {-B, C}]
    compare_models(KB)

if __name__ == '__main__':
    main()
```

3. Output

3.1. Model output

```
---- START DPLL ----
knowledge base clauses: [{A, B}, {-C, A}, {D, -A, B}]
symbols: [D, -C, A, B]
Now recursing ...
- clause statuses: ['undetermined', 'undetermined', 'undetermined']
- model: {}
- symbols left: [D, -C, A, B]

Degree heuristic chooses next symbol: A
- clause statuses: ['true', 'true', 'undetermined']
- model: {A: 'true'}
- symbols left: [D, -C, B]

Pure symbol heuristic chooses next literal: B
- clause statuses: ['true', 'true', 'true']
- model: {A: 'true', B: 'true'}
- symbols left: [D, -C]

------ RESULT ------
satisfiable: True
model: {A: 'true', B: 'true', D: 'free', -C: 'free'}


---- START DPLL ----
knowledge base clauses: [{-A, B, E}, {A, -B}, {A, -E}, {D, -E}, {-C, -B, -F}, {B, -E},
↪  {-B, F}, {C, -B}]
symbols: [B, -F, D, E, -C, -A]
Now recursing ...
- clause statuses: ['undetermined', 'undetermined', 'undetermined', 'undetermined',
↪  'undetermined', 'undetermined', 'undetermined', 'undetermined']
- model: {}
- symbols left: [B, -F, D, E, -C, -A]

Degree heuristic chooses next symbol: B
- clause statuses: ['true', 'undetermined', 'undetermined', 'undetermined',
↪  'undetermined', 'true', 'undetermined', 'undetermined']
- model: {B: 'true'}
- symbols left: [-F, D, E, -C, -A]

Unit clause heuristic chooses next literal: A
- clause statuses: ['true', 'true', 'true', 'undetermined', 'undetermined', 'true',
↪  'undetermined', 'undetermined']
- model: {B: 'true', A: 'true'}
- symbols left: [-F, D, E, -C]

Unit clause heuristic chooses next literal: F
- clause statuses: ['true', 'true', 'true', 'undetermined', 'undetermined', 'true',
↪  'true', 'undetermined']
- model: {B: 'true', A: 'true', F: 'true'}
- symbols left: [D, E, -C]

Unit clause heuristic chooses next literal: -C
```

```
- clause statuses: ['true', 'true', 'true', 'undetermined', 'false', 'true', 'true',
↪  'true']
- model: {B: 'true', A: 'true', F: 'true', -C: 'true'}
- symbols left: [D, E]

- clause statuses: ['true', 'true', 'true', 'undetermined', 'true', 'true', 'true',
↪  'false']
- model: {B: 'true', A: 'true', F: 'true', -C: 'false'}
- symbols left: [D, E]

- clause statuses: ['true', 'true', 'true', 'undetermined', 'true', 'true', 'false',
↪  'undetermined']
- model: {B: 'true', A: 'true', F: 'false'}
- symbols left: [D, E, -C]

- clause statuses: ['true', 'false', 'undetermined', 'undetermined', 'undetermined',
↪  'true', 'undetermined', 'undetermined']
- model: {B: 'true', A: 'false'}
- symbols left: [-F, D, E, -C]

- clause statuses: ['undetermined', 'true', 'undetermined', 'undetermined', 'true',
↪  'undetermined', 'true', 'true']
- model: {B: 'false'}
- symbols left: [-F, D, E, -C, -A]

Unit clause heuristic chooses next literal: -E
- clause statuses: ['true', 'true', 'undetermined', 'undetermined', 'true', 'false',
↪  'true', 'true']
- model: {B: 'false', -E: 'true'}
- symbols left: [-F, D, -C, -A]

- clause statuses: ['undetermined', 'true', 'true', 'true', 'true', 'true', 'true',
↪  'true']
- model: {B: 'false', -E: 'false'}
- symbols left: [-F, D, -C, -A]

Unit clause heuristic chooses next literal: -A
- clause statuses: ['false', 'true', 'true', 'true', 'true', 'true', 'true', 'true']
- model: {B: 'false', -E: 'false', -A: 'true'}
- symbols left: [-F, D, -C]

- clause statuses: ['true', 'true', 'true', 'true', 'true', 'true', 'true', 'true']
- model: {B: 'false', -E: 'false', -A: 'false'}
- symbols left: [-F, D, -C]

------ RESULT ------
satisfiable: True
model: {B: 'false', -E: 'false', -A: 'false', -F: 'free', D: 'free', -C: 'free'}
```

3.2. Comparison output

```
Comparing models ...
Without heuristics: mean = 0.00812, std = 0.00318
With heuristics: mean = 0.00368, std = 0.00021
[Finished in 1.452s]
```