CS152 Assignment 2: DPLL Algorithm

Yoav Rabinovich, Dec 2018

Sadly, I managed to break this implementation while working. I find it hard to pinpoint my mistake, but the testing isn't working. I unfourtunately ran out of time and 24-hour extensions trying to fix this implementation. I hope the documentation is clear enough to infer my level of understanding. I believe I have implemented the extensions successfully, and that the problem lies somewhere in the first few lines of the DPLL function.

Apologies, Yoav.

Link to notebook: https://gist.github.com/Shesh6/28a5957e37b3da25bef2f121c68dbd69#file-cs152assignment2-ipynb

Q1: Literal Class

```
class Literal(object):
  """
  This class represents a literal, and holds a name and a sign.
  A negative sign is assigned by negating the object.
  Equality is defined simply by equality of name,
  since only a negated literal is false.
  """

  def __init__(self, name):
    self.name = name
    self.sign = True

  def flipSign(self):
    self.sign = not self.sign

  # Hint 3: Using one set for literals
  def __eq__(self, other):
    return self.name == other.name

  def __neg__(self):
    neg = Literal(self.name)
    neg.flipSign()
    return neg

  # Hash and repr, for use in dictionaries
  def __hash__(self):
    return hash(self.name)

  def __str__(self):
    return self.name

  def __repr__(self):
    return str(self)
```

Q2: DPLL Implementation

```
#Q2: DPLL Implementation

# Hint 4: Global variable for model
model={}

def DPLL_Satisfiable(KB):
```

```
  '''
  Takes a KB in CNF form and runs DPLL.
  Returns True if the KB is satisfiable.
  The model isn't returned because it's a global variable.
  '''

  global model

  # Uses a set to list all unique symbols in the KB
  symbols = set()
  for clause in KB:
    for s in clause:
      symbols.add(s)
  symbols = list(symbols)
  # Call DPLL with the list of symbols
  satisfiable = DPLL(KB, symbols)

  # Assign free variables
  if satisfiable==True:
    for s in symbols:
      if s not in model:
        model[s] = "free"

  return satisfiable
```

```
def DPLL(clauses, symbols):
  '''
  DPLL Algorithm
  '''

  global model
#   print("call")
#   print("model: " + str(model))

  # Variable for tracking if the model is satisfying
  model_sat = True
  # Placeholder for the KB to be fed into the next recursive iteration
  next_KB = []

  # Iterate over all clauses
  for i, c in enumerate(clauses):
#     print("clause: " + str(c))
    # Variable for tracking if the clause is true (Hint 2)
    done = False
    # Counting all false literals (Hints 1,3)
    falses = set()

    # Hint 2: One true literal in the model confirms true clause
    for s in c:
      if s in model:
        if model[s] == s.sign:
#           print("confirmed clause " + str(c) + " by symbol " + str(s))
          done = True
          break
        else:
#           print (str(s)+" is false, adding")
          falses.add(s)

    # Hint 1: False literals can be dropped
    if not done:

      # Model disproven
#       print("model disproven")
      model_sat = False

      # Shorten clause
      # clauses[i] = [x for x in c if s not in falses]
      clauses[i] = set(c) ^ falses
#       print("shortened clause to " + str(clauses[i]))

      # Add the shortened clause to the next iteration KB
      next_KB.append(clauses[i])

```

```
      # Termination Condition 1: A clause is false
      # since all literals were false
      if len(clauses[i]) == 0:
#         print("cond 1, all literals false")
        return False

  # Termination Condition 2: Model wasn't disproven
  if model_sat:
#     print("cond 2, model wasn't disproven")
    return True

  models = []

  # Pure symbol heuristic, or unit clause heuristic
#   print("trying pure symbol")
  p, value = pure_symbol_heuristic(symbols, clauses)
  if p is None:
#     print("trying unit clause")
    p, value = unit_clause_heuristic(clauses)

  if p is not None:
    rest = symbols.copy()
    rest.remove(p)
    next_m = model.copy()
    # Set up next model to be tested
    next_m[p] = value
    models.append(next_m)

  # If all else fails, use first heuristic
  else:
#     print("resorting to first heuristic")
    p, rest = symbols[0], symbols[1:]
    next_m_true = model.copy()
    next_m_true[p] = True
    next_m_false = model.copy()
    next_m_false[p] = False
    models.append(next_m_true)
    models.append(next_m_false)

#   print("prepared to test " + str(models))
  # Call DPLL recursively to try our models
  for m in models:
    model = m
#     print("testing " + str(model))
#     print("with KB" + str(next_KB))
#     print("and symbols " + str(rest))
    satisfiable = DPLL(next_KB, rest)

    # Termination propagation
    if satisfiable==True:
#       print("cond 4, propagation")
      return True, model

  # Termination Condition 3: Satisfiability wasn't inferred
#   print("cond 3, nope")
  return False
```

E1: Degree Heuristic

```
from collections import defaultdict

def degree_heuristic(symbols, clauses):
  '''
  Separate most frequently appearing literal from the rest
  '''

  # Defaultdict for counting by literal name
  count = defaultdict(int)
  high_c = 0
  allowed_symbols = set(symbols)
```

```
  high_s = None

  # Iterate over symbols in KB and keep track of frequency
  for c in clauses:
    for s in c:
      count[s] += 1
      if count[s] > high_c:
        if s in set(symbols):
          high_c = count[s]
          high_s = s
  rest = symbols.copy()
  rest.remove(high_s)
  return high_s, rest
```

E2: Pure Symbol and Unit Clause Heuristics:

```
def pure_symbol_heuristic(symbols, clauses):
  '''
  Returns the most common pure symbol.
  (A symbol that occurs with the same sign throught the KB)
  '''

  global model

  pures = set(symbols)
  values = {}

  # Iterate through symbols and keep track of their values
  for c in clauses:
    for s in c:
      if s not in values:
        values[s] = s.sign

      # If a symbol is found with the wrong value, it's not pure
      if values[s] != s.sign:
        if s in pures:
          pures.remove(s)

  # Extra Kudos: Combine with degree heuristic
  if len(pures) >= 1:
    p, value = degree_heuristic(pures, clauses=clauses)
    value = values[p]
    # Check that unit clause doen't contradict the model
    if p in model:
      if model[p] == value:
        # Return the unit clause symbol
        return s, s.sign

  # If none found, return none
  return None, None
```

```
def unit_clause_heuristic(clauses):
  '''
  Returns a symbol that occurs in a unit clause.
  (clause with only one literal)
  '''

  global model

  for c in clauses:
    # Look for unit clause
    if len(c) == 1:
      # Check that unit clause doen't contradict the model
      s = next(iter(c))
      if s in model:
        if model[s] == s.sign:
          # Return the unit clause symbol
          return s, s.sign
  # If none found, return none
```

```
    return None, None
```

Q3: Test with Exercise 7.20 of Russell & Norvig:

Converstion to CNF:

1. A <-> (BvE)
   A -> (BvE)^(BvE) -> A (Bidirectional implication elimination)
   (-AvBvE))^(-(BvE)vA) (Implication elimination)
   (-AvBvE)^((-B^-E)vA) (De Morgens)
   (-AvBvE)^(-BvA)^(-EvA)) (Distributivity)
2. E -> D
   -EvD (Implication elimination)
3. C^F -> -B
   -(C^F)v-B (Implication elimination)
   -Cv-Fv-B (De Morgens)
4. E -> B
   -EvB (Implication elimination)
5. B -> F
   -BvF (Implication elimination)
6. B->C
   -BvC (Implication elimination)

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

clauses = [{-A, B, E},{-B, A},{-E, A},{-E, D},
           {-C, -F, -B},{-E, B},{-B, F},{-B, C}]
model={}
satisfiable = DPLL_Satisfiable(clauses)
print(satisfiable, model)
```