ENSTA Paris
2022-2023

Logical models for Artificial Intelligence – INF656L

Alexandre Chapoutot

SAT Part - Practical work 2

Goal(s)

★ Implementation of WalkSAT algorithm

Exercise 1 – The Labyrinth Guardians.

You are walking in a labyrinth and all of a sudden you find yourself in front of three possible roads: the road on your left is paved with gold, the one in front of you is paved with marble, while the one on your right is made of small stones. Each street is protected by a guardian. You talk to the guardians and this is what they tell you:

- *The guardian of the gold street*: "This road will bring you straight to the center. Moreover, if the stones take you to the center, then also the marble takes you to the center."
- *The guardian of the marble street*: "Neither the gold nor the stones will take you to the center."
- *The guardian of the stone street*: "Follow the gold and you'll reach the center, follow the marble and you will be lost."

Given that you know that all the guardians are liars, can you choose a road being sure that it will lead you to the center of the labyrinth? If this is the case, which road you choose?

Solution:
Language

- g: "the gold road brings to the center"
- m: "the marble road brings to the center"
- s: "the stone road brings to the center"

Propositional encoding

- **The guardian of the gold street is a liar**

$$\neg(g \wedge (s \implies m)) \quad \text{simplied in} \quad \neg g \vee (s \wedge \neg m) \tag{1}$$

- **The guardian of the marble street is a liar**

$$\neg(\neg g \wedge \neg s) \quad \text{simplied in} \quad g \vee s \tag{2}$$

- **The guardian of the stone street is a liar**

$$\neg(g \wedge \neg m) \quad \text{simplied in} \quad \neg g \vee m \tag{3}$$

Solution

g	m	s	$(1) \wedge (2) \wedge (3)$
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	0
0	1	**1**	**1**
0	1	0	0
0	0	**1**	**1**
0	0	0	0

We have two possible interpretations that satisfy the propositions, and in both of them the stone street brings to the center. Thus I can choose the stone street being sure that it leads to the center.

Exercise 2 – WalkSAT Algorithm

We will use a simple representation of the Boolean constraints in CNF. Specifically, we will consider a data structure of list of integer lists as in the case of the TD1.

```
function WALKSAT(clauses, p, max-flips) returns a satisfying model or failure
   inputs: clauses, a set of clauses in propositional logic
              p, the probability of choosing to do a "random walk" move, typically
around 0.5
              max-flips, number of flips allowed before giving up
   model ← a random assignment of true/false to the symbols in clauses
   for i = 1 to max-flips do
      if model satisfies clauses then return model
      clause ← a randomly selected clause from clauses that is false in model
      with probability p flip the value in model of a randomly selected symbol
from clause
      else flip whichever symbol in clause maximizes the number of satisfied clauses
   return failure
```

Figure 1: Pseudo code WalkSAT

Question 1

Inspired by the pseudo-code[1] given to Figure , implement this algorithm (*e.g.*, in Python)

[1] Image coming from *Artificial Intelligence: A Modern Approach*

To test your solver some problems in DIMACS format can be found on

`https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

Solution:

```python
import random as ran

def from_file(inputfile):
    clauses = list()
    VARS = set()
    p = 0
    cnf = 0
    with open(inputfile, "r") as input_file:
        clauses.append(list())
        maxvar = 0
        for line in input_file:
            tokens = line.split()
            if len(tokens) != 0 and tokens[0] not in ("p", "c"):
                for tok in tokens:
                    lit = int(tok)
                    maxvar = max(maxvar, abs(lit))
                    if lit == 0:
                        clauses.append(list())
                    else:
                        clauses[-1].append(lit)
            if tokens[0] == "p":
                p = int(tokens[2])
                cnf = int(tokens[3])
        assert len(clauses[-1]) == 0
        clauses.pop()
        if (maxvar > p):
            print("Non-standard CNF encoding!")
            sys.exit(5)
    # Variables are numbered from 1 to p
    for i in range(1, p + 1):
        VARS.add(i)
    return VARS, clauses


def bitfield(n):
    return [True if digit=='1' else False for digit in bin(n)[2:]] # [2:] to chop off the "0b" part

def generateOneModel (n, i):
    w = bitfield(i)
    rest = n - len(w)
    wr = ([False] * rest) + w
    return wr

def generateAllModel (n):
    allValues = []
    for i in range(0, pow(2, n)):
        wr = generateOneModel (n, i)
        allValues.append(wr)
    return allValues

def evalCNF (cnf, model):
    formulaValue = True
    trueClauses = []
    falseClauses = []
    for clause in cnf:
        clauseValue = False
        for lit in clause:
            if (lit > 0):
                clauseValue = clauseValue or model[lit-1]
            else:
                clauseValue = clauseValue or (not model[(-lit)-1])
        formulaValue = formulaValue and clauseValue
        if clauseValue:
            trueClauses.append (clause)
        else:
            falseClauses.append (clause)
    return formulaValue, trueClauses, falseClauses

def getVars (clause):
    varList = []
    for lit in clause:
        if (lit < 0):
           varList.append ((-lit)-1) # check if we generate proper set of indices
        else:
            varList.append (lit-1)
    return list(set(varList)) # unicity of variable names

def maximizeTrueClause (cnf, model, fc):
    nbTrueClauses = 0
    newModel = model
    for i in getVars (fc):
        temp = model
        temp[i] = not (temp[i])
        ans, tc, fc = evalCNF (cnf, model)
        if (len(tc) > nbTrueClauses):
            nbTrueClauses = len(tc)
            newModel = temp
    return newModel

def walkSAT (nbVar, cnf, p, maxflips):
    model = generateOneModel (0, pow(2, nbVar))
    for i in range (maxflips):
        (tVal, tClauses, fClauses) = evalCNF(cnf, model)
        if tVal:
            return True, i, model
        fc = ran.choice(fClauses)
        if ran.random() >= p:
            # flip the value in model of a randomly selected symbol from fc
            index = ran.randrange (0, len(fc))
            model[index] = not (model[index])
        else:
            # flip whichever symbol in fc maximizes the number of satisfied clauses
            model = maximizeTrueClause (cnf, model, fc)
    return False, maxflips, []
        ###

def decisionProcedure (nbVar, cnf):
    (ans, cpt, m) = walkSAT (nbVar, cnf, 0.9, 100000)
    if (ans):
        print ("SAT(", cpt, ") with ", m)
    else:
        print ("Can't conclude on satisfiability")
```

A Programming help

Python's SymPy library provides an implementation of the DPLL/CDCL algorithm that you may find useful in verifying your WalkSAT implementation.

For example, the Python source code uses the DPLL algorithm in Function satisfiable

```
from sympy.logic.boolalg import And, Or, Implies, Equivalent, Not, to_cnf
from sympy.abc import p, q, r
from sympy.logic.inference import satisfiable

expr = Implies(p, Equivalent (q, r))
print(expr)

expr_cnf = to_cnf(expr)
print(expr_cnf)

print(satisfiable(expr_cnf))

from sympy.logic.utilities.dimacs import load
expr2_cnf = load('1 2 \n 3')
print(expr2_cnf)
print(satisfiable(expr2_cnf))
```

