# hw02

February 2, 2023

## 1 CS 4100 Homework 02

**Due Thursday 2/2 at midnight (1 minute after 11:59 pm) in Gradescope (with a grace period of 6 hours)**

**You may submit the homework up to 24 hours late (with the same grace period) for a penalty of 10%.** You must submit the homework in Gradescope as a zip file containing two files: - The .ipynb file (be sure to Kernel -> Restart and Run All before you submit); and - A .pdf file of the notebook.

For best results obtaining a clean PDF file on the Mac, select File -> Print Review from the Jupyter window, then choose File-> Print in your browser and then Save as PDF. Something similar should be possible on a Windows machine.

All homeworks will be scored with a maximum of 100 points; if point values are not given for individual problems, then all problems will be counted equally.

Note: I have uploaded my solution for HW 01 on the class web page if you wish to use it; some of the code in HW 01 will be needed in this homework.

[1]:
```
# My solution from HW01
import math
from itertools import product

def evalLiteral(I,L):
    if I[abs(L)-1]:
        return L > 0
    else:
        return (-1 * L) > 0

def evalClause(I,C):
    return any([evalLiteral(I, i) for i in C])

def evalCNF(I,A):
    return all([evalClause(I, i) for i in A])

# Prints out the interpretations of a CNF statement;
# takes in a list of list of integers (cnf statement with clauses)
# and an integer (count of literals)
```

```
def proveCNF(A, N):
    satlist = [i*evalCNF(list(i), A) for i in list(product([True,␣
 ↪False],repeat=N))]
    print([i*evalCNF(list(i), A) for i in list(product([True,␣
 ↪False],repeat=N))])
    if all(i == () for i in satlist):
        print("Unsatisfiable")
    elif not any(i == () for i in satlist):
        print("True")
    else:
        print("Satisfiable")
    print("There are "+str(len([x for x in satlist if x != ()]))+" satisfiable␣
 ↪interpretations.")
```

Here are some useful utility functions for examining the results of your code.

[2]:
```
# This will pretty-print literals, clauses, and clause lists.
# It will print out propositional letters as A, B, C, ...

def literal2String(L):
    return ('-' if (L<0) else '') + chr(ord('A')+abs(L)-1)

def clause2String(C):
    if(C=={}):
        return '{}'
    else:
        return '{ '+(', '.join([literal2String(L) for L in C]))+' }'

def clauseList2String(A):
    return '[ '+(', '.join([clause2String(C) for C in A]))+' ]'

def pprint(X):
    if(type(X)==int):               # X is literal
        print(literal2String(X))
    elif(type(X)==set):             # X is a clause
        print(clause2String(X))
    elif(type(X)==list):            # X is a clause list
        print(clauseList2String(X))
    else:
        print('Error in pprint!')

        #test

pprint(-2)
pprint({3,-2})
pprint([ {1},{-1,2}, {-2,-1,3} ])
pprint([ {1}, {}, {-2} ])
```

```
-B
{ C, -B }
[ { A }, { B, -A }, { C, -B, -A } ]
[ { A }, {}, { -B } ]
```

## 1.1 Problem One: Resolution Theorem Proving in Propositional Logic (30 pts)

In this problem we will explore the resolution calculus, using breadth-first search to find the empty clause (if it exists) in the resolvents generated from the input set.

### 1.1.1 Part A: Resolution Rule

Complete the following template according to the definition of resolution given in your text and in lecture.

[3]:
```
from itertools import accumulate

# Return all possible resolvents of clauses C1 and C2 (sets)

def resolve(C1,C2):
    C1o = C1.copy()
    C2o = C2.copy()
    soln = []
    for i in C1o:
        if -1*i in C2o:
            C1.remove(i)
            C2.remove(-1*i)
            soln.append(C1|C2)
            C1 = C1o.copy()
            C2 = C2o.copy()
    return soln               # just to get it to compile, replace with your␣
 ↪code


# Tests -- Note that your sets may occur in a different order than shown here

C1 = {1}
C2 = {-1}
print(clause2String(C1),clause2String(C2))
print(clauseList2String(resolve(C1,C2)),'\n')   # should be [ {} ]

C1 = {1,2}
C2 = {-1,-3}
print(clause2String(C1),clause2String(C2))
print(clauseList2String(resolve(C1,C2)),'\n')   # should be [ { B, -C } ]

C1 = {}
C2 = {1,-2}
```

```
print(clause2String(C1),clause2String(C2))
print(clauseList2String(resolve(C1,C2)),'\n')   # should be [  ]

C1 = {1,-3}
C2 = {1,-2}
print(clause2String(C1),clause2String(C2))
print(clauseList2String(resolve(C1,C2)),'\n')   # should be [  ]

C1 = {1,2,3}
C2 = {-1,-2,-3}
print(clause2String(C1),clause2String(C2))
print(clauseList2String(resolve(C1,C2)),'\n')   # should be [ { B, C, -C, -B },␣
 ↪{ A, C, -C, -A }, { A, B, -A, -B } ]
```

```
{ A } { -A }
[ {  } ]

{ A, B } { -C, -A }
[ { B, -C } ]

{} { A, -B }
[  ]

{ A, -C } { A, -B }
[  ]

{ A, B, C } { -C, -A, -B }
[ { B, C, -C, -B }, { A, C, -C, -A }, { A, B, -A, -B } ]
```

### 1.1.2 Part B: Resolution Rule Continued

Complete the following template according to the definition of resolution given in your text and in lecture.

[4]:
```
# A is a list of clauses, C is a clause
# Return list of all possible resolvents of C with a clause in A

def resolveAll(A,C):
    Co = C.copy()
    soln = []
    for x in range(len(A)):
        newC = A[x].copy()
        soln += resolve(C, newC)
        C = Co.copy()
    return soln

# tests
```

```
KB = [ {-2,-1,3} ]
NQ = {-3}

pprint(KB)
pprint(NQ)
pprint(resolveAll(KB,NQ))     # should be [ { -B, -A } ]
print()

KB = [ {1}, {}, {-2,-1,-3} ]
NQ = {-3}

pprint(KB)
pprint(NQ)
pprint(resolveAll(KB,NQ))     # should be [  ]
print()

KB = [ ]
NQ = {-3}

pprint(KB)
pprint(NQ)
pprint(resolveAll(KB,NQ))     # should be [  ]
print()

KB = [ {1}, {-1,2}, {-2,-1,3} ]
NQ = {-3,1}

pprint(KB)
pprint(NQ)
pprint(resolveAll(KB,NQ))     # should be [ { B, -C }, { A, -B, -A }, { C, -C,
 ↪-B } ]
```

```
[ { C, -B, -A } ]
{ -C }
[ { -B, -A } ]

[ { A }, {}, { -C, -B, -A } ]
{ -C }
[  ]

[  ]
{ -C }
[  ]

[ { A }, { B, -A }, { C, -B, -A } ]
{ A, -C }
[ { B, -C }, { C, -C, -B }, { A, -B, -A } ]
```

### 1.1.3 Part C: Breadth-first Prover

Write a function `prove1` which takes a set `CL` and searches for the empty clause by inserting all members of CL into a queue, and generating resolvents from the clause popped off the front of the queue with any member of the rest of queue, and adding them to the back of the queue. It should behave as shown below:

- If the empty clause is found, report "Unsatisfiable";
- If the queue empties out, report "Satisfiable";
- If the prover can not find the empty clause after 'limit' steps, report that fact.

In each case, you should also report the number of steps (= times you popped the front off the queue);

If the parameter `trace` is true, you should print out the queue at each step.

[5]:
```
# perform breath-first search from a set of clauses CL, by
# inserting all of CL into a queue
# and then searching using BFS by generating resolvents of head of queue with␣
 ↪any
# clause in the queue

def prove1(CL,limit=30,trace=True):
    queue = CL.copy()
    for i in range(limit):
        if trace:
            pprint(queue)
        if queue == []:
            print("Satisfiable")
            print(str(i)+" steps were taken")
            return True
        curr = queue.pop(0)
        queue2 = queue.copy()
        for y in resolveAll(queue2, curr):
            if y == set():
                print("Unsatisfiable")
                print(str(i+1)+" steps were taken")
                return False
            else:
                queue.append(y)

    print("The empty clause cannot be found after "+str(limit)+" steps.")
    return False
```

[6]:
```
# tests: first 3 are unsatisfiable, test 4 is satisfiable

print('\ntest a','\n------')
CL1a = [ {2}, {-2} ]
```

```
print('CL1a: ',end=''); pprint(CL1a); print()
prove1(CL1a); print()

print('test b','\n------')
CL1b = [ {1}, {-1,2}, {-2,3}, {-3,4}, {-4} ]

print('CL1b: ',end=''); pprint(CL1b); print()
prove1(CL1b); print()

print('test c','\n------')
CL1c = [ {1}, {-1,2,3}, {-3,4}, {-3,5}, {-5}, {-2} ]

print('CL1c: ',end=''); pprint(CL1c); print()
prove1(CL1c); print()

print('test d','\n------')
CL1d = [ {1}, {-1,2,3}, {-3,4}, {-3,5}, {-5} ]

print('CL1d: ',end=''); pprint(CL1d); print()
prove1(CL1d); print()
```

```
test a
------
CL1a: [ { B }, { -B } ]

[ { B }, { -B } ]
Unsatisfiable
1 steps were taken

test b
------
CL1b: [ { A }, { B, -A }, { C, -B }, { D, -C }, { -D } ]

[ { A }, { B, -A }, { C, -B }, { D, -C }, { -D } ]
[ { B, -A }, { C, -B }, { D, -C }, { -D }, { B } ]
[ { C, -B }, { D, -C }, { -D }, { B }, { C, -A } ]
[ { D, -C }, { -D }, { B }, { C, -A }, { D, -B }, { C } ]
[ { -D }, { B }, { C, -A }, { D, -B }, { C }, { -C }, { D, -A }, { D } ]
Unsatisfiable
5 steps were taken

test c
------
CL1c: [ { A }, { B, C, -A }, { D, -C }, { E, -C }, { -E }, { -B } ]

[ { A }, { B, C, -A }, { D, -C }, { E, -C }, { -E }, { -B } ]
```

```
[ { B, C, -A }, { D, -C }, { E, -C }, { -E }, { -B }, { B, C } ]
[ { D, -C }, { E, -C }, { -E }, { -B }, { B, C }, { B, D, -A }, { B, E, -A }, {
C, -A } ]
[ { E, -C }, { -E }, { -B }, { B, C }, { B, D, -A }, { B, E, -A }, { C, -A }, {
B, D }, { D, -A } ]
[ { -E }, { -B }, { B, C }, { B, D, -A }, { B, E, -A }, { C, -A }, { B, D }, {
D, -A }, { -C }, { B, E }, { E, -A } ]
[ { -B }, { B, C }, { B, D, -A }, { B, E, -A }, { C, -A }, { B, D }, { D, -A },
{ -C }, { B, E }, { E, -A }, { B, -A }, { B }, { -A } ]
Unsatisfiable
6 steps were taken

test d
------
CL1d: [ { A }, { B, C, -A }, { D, -C }, { E, -C }, { -E } ]

[ { A }, { B, C, -A }, { D, -C }, { E, -C }, { -E } ]
[ { B, C, -A }, { D, -C }, { E, -C }, { -E }, { B, C } ]
[ { D, -C }, { E, -C }, { -E }, { B, C }, { B, D, -A }, { B, E, -A } ]
[ { E, -C }, { -E }, { B, C }, { B, D, -A }, { B, E, -A }, { B, D } ]
[ { -E }, { B, C }, { B, D, -A }, { B, E, -A }, { B, D }, { -C }, { B, E } ]
[ { B, C }, { B, D, -A }, { B, E, -A }, { B, D }, { -C }, { B, E }, { B, -A }, {
B } ]
[ { B, D, -A }, { B, E, -A }, { B, D }, { -C }, { B, E }, { B, -A }, { B }, { B
} ]
[ { B, E, -A }, { B, D }, { -C }, { B, E }, { B, -A }, { B }, { B } ]
[ { B, D }, { -C }, { B, E }, { B, -A }, { B }, { B } ]
[ { -C }, { B, E }, { B, -A }, { B }, { B } ]
[ { B, E }, { B, -A }, { B }, { B } ]
[ { B, -A }, { B }, { B } ]
[ { B }, { B } ]
[ { B } ]
[ ]
Satisfiable
14 steps were taken
```

## 1.2 Problem Two (10 pts)

In this problem, we will experiment with our prover to understand its behavior.

### 1.2.1 Part A

Generate a list of clauses `CL2a` such that

- CL2a is satisfiable, but
- It will make the prover run forever (in this case, not terminating with "Satisfiable!" within 30 steps)

You must

- Pretty-print CL2a
- Prove that it is satisfiable (using your code from HW 01),
- Show that it fails to terminate after 30 steps, and
- Explain why your example behaves in this way.

Note: Your example should be such that no matter how large `limit` is, it will not terminate; however, for purposes of grading, you should leave `limit=30` but `trace=False`.

Hint: See if you can get the prover to loop uselessly on the same clauses. It may help to think about the order in which clauses will be processed.

[7]:
```
print('test 2a','\n------')
CL2a = [{-1, 2}, {-2, 1}, {-2, 1}]

proveCNF(CL2a, 2)
print()

print('CL2a: ',end=''); pprint(CL2a); print()
prove1(CL2a, 30, False); print()

# This clause contains statements that logically imply each other,
# causing an infinite loop where the same clauses are generated multiple times.
```

```
test 2a
------
[(True, True), (), (), (False, False)]
Satisfiable
There are 2 satisfiable interpretations.

CL2a: [ { B, -A }, { A, -B }, { A, -B } ]

The empty clause cannot be found after 30 steps.
```

### 1.2.2 Part B

In this problem we will explore what happens when the clause set causes an exponential explosion in the search space. We will use a classic example of a clause set which requires exponential time (assuming $P \neq NP$).

First create a clause list `CL2b` from only 2 symbols which is unsatisfiable but causes exponential behavior. The easiest way to do this is to create a clause set with 4 clauses, each of which makes one row of the truth table for two symbols (say A and B) false. For example, for a row that has **(True,False)**, you would create $\{\neg A, B\} = \neg A \vee B$. If you make sure that no row of the truth table is a model, then it must be unsatisfiable, but each clause is necessary to prove this fact (if you leave a clause out, then that row of the table would be a model).

It should take 7 steps to determine unsatisfiability.

As in Part A,

- Pretty-print CL2b
- Verify that it is unsatisfiable using code from HW 01
- Show that it takes 7 steps to find the empty clause.

[8]:
```
print('test 2b','\n------')
CL2b = [ {-1, 2}, {1, 2}, {-1, -2}, {1, -2} ]

proveCNF(CL2b, 2)
print()

print('CL2b: ',end=''); pprint(CL2b); print()
prove1(CL2b); print()
```

```
test 2b
------
[(), (), (), ()]
Unsatisfiable
There are 0 satisfiable interpretations.

CL2b: [ { B, -A }, { A, B }, { -A, -B }, { A, -B } ]

[ { B, -A }, { A, B }, { -A, -B }, { A, -B } ]
[ { A, B }, { -A, -B }, { A, -B }, { B }, { -A }, { A, -A }, { B, -B } ]
[ { -A, -B }, { A, -B }, { B }, { -A }, { A, -A }, { B, -B }, { B, -B }, { A, -A
}, { A }, { B }, { A, B }, { A, B } ]
[ { A, -B }, { B }, { -A }, { A, -A }, { B, -B }, { B, -B }, { A, -A }, { A }, {
B }, { A, B }, { A, B }, { -B }, { -A }, { -B, -A }, { -A, -B }, { -A, -B }, {
-B, -A }, { -B }, { -A }, { B, -B }, { A, -A }, { B, -B }, { A, -A } ]
[ { B }, { -A }, { A, -A }, { B, -B }, { B, -B }, { A, -A }, { A }, { B }, { A,
B }, { A, B }, { -B }, { -A }, { -B, -A }, { -A, -B }, { -A, -B }, { -B, -A }, {
-B }, { -A }, { B, -B }, { A, -A }, { B, -B }, { A, -A }, { A }, { -B }, { A, -B
}, { A, -B }, { A, -B }, { A, -B }, { A }, { A }, { A }, { -B }, { -B }, { -B },
{ -B }, { -B }, { -B }, { A, -B }, { A, -B }, { A, -B }, { A, -B } ]
Unsatisfiable
5 steps were taken
```

### 1.2.3 Part C

Now, create a clause list `CL2d`from 3 symbols, with 8 clauses, along the same lines as the previous example (each clause makes one line of the truth table false).

To do:

- Pretty-print CL2d
- Prove that it is unsatisfiable as above
- Run this with a limit of 3, 4, and 5 to get a sense for what this will do. Then turn off tracing and try it for a few more (very soon it will simply get lost in an exponential explosion).

- Answer this question: How large could you make limit and still get "Empty clause not found after …" to print out within say a minute?

[9]:
```
print('test 2d','\n------')
CL2d = [ {1, 2, 3}, {-1,2,3}, {1, 2, -3}, {1, -2, -3}, {1, -2, 3}, {-1, 2, -3},␣
 ↪{-1, -2, -3}, {-1, -2, 3} ]

proveCNF(CL2d, 3)
print()

print('CL2d: ',end=''); pprint(CL2d); print()
prove1(CL2d, 13, False); print()

# For my machine, 13 steps is the most it can process in a minute
```

```
test 2d
------
[(), (), (), (), (), (), (), ()]
Unsatisfiable
There are 0 satisfiable interpretations.

CL2d: [ { A, B, C }, { B, C, -A }, { A, B, -C }, { A, -C, -B }, { A, C, -B }, {
B, -C, -A }, { -C, -A, -B }, { C, -A, -B } ]

The empty clause cannot be found after 13 steps.
```

## 1.3 Refinements of Resolution

There are various ways to make resolution more efficient, including

- Set of Support Strategy: partition your clause set into subsets KB and SOS, such that KB is satisfiable, but KB + SOS is unsatisfiable;
- Removing useless clauses such as tautologies and duplicates
- Shortest-Clauses-First: use a priority queue instead of a queue, and keep it ordered by size, with smaller clauses to the front.

The motivation for all of these is to prevent doing useless work, and–in the case of the shortest-first–to try to focus on creating short clauses as soon as possible.

## 1.4 Problem 3 (Set of Support) (10 pts)

### 1.4.1 Part A

Rewrite your prover to create a function `prove3` which uses the set of support strategy, where the set of unsatisfiable clauses is separated into KB and SOS, in such a way that KB is known to be satisfiable.

- Your prover will take KB and SOS as separate arguments;
- Create your initial queue from SOS alone;

- In each step, remove the head of the queue, and form resolvents between that clause and any clause in KB + Queue.

Rewrite the tests from Problem One, Part A, such that the *last* clause in each list is removed and put in the SOS. The first one is done for you to show what is needed.

Run your prover on these four examples to verify that all is working as expected.

[10]:
```
def prove3(KB, SOS, limit=30,trace=True):
    queue = SOS.copy()
    for i in range(limit):
        if trace:
            pprint(queue)
        if queue == []:
            print("Satisfiable")
            print(str(i)+" steps were taken")
            return True
        curr = queue.pop(0)
        queue2 = KB.copy() + queue.copy()
        for y in resolveAll(queue2, curr):
            if y == set():
                print("Unsatisfiable")
                print(str(i+1)+" steps were taken")
                return False
            else:
                queue.append(y)

    print("The empty clause cannot be found after "+str(limit)+" steps.")
    return False
```

[11]:
```
print('\ntest 1','\n------')
KB3a1 = [ {2} ]
SOS3a1 = [ {-2} ]

print('KB3a1: ',end=''); pprint(KB3a1);
print('SOS3a1: ',end=''); pprint(SOS3a1); print()
prove3(KB3a1,SOS3a1); print()

# etc for other 3

print('\ntest 2','\n------')
KB3a2 = [ {1}, {-1,2}, {-2,3}, {-3,4} ]
SOS3a2 = [ {-4} ]

print('KB3a2: ',end=''); pprint(KB3a2);
print('SOS3a2: ',end=''); pprint(SOS3a2); print()
prove3(KB3a2,SOS3a2); print()
```

```
print('\ntest 3','\n------')
KB3a3 = [ {1}, {-1,2,3}, {-3,4}, {-3,5}, {-5} ]
SOS3a3 = [ {-2} ]

print('KB3a3: ',end=''); pprint(KB3a3);
print('SOS3a3: ',end=''); pprint(SOS3a3); print()
prove3(KB3a3,SOS3a3); print()

print('\ntest 4','\n------')
KB3a4 = [ {1}, {-1,2,3}, {-3,4}, {-3,5} ]
SOS3a4 = [ {-5} ]

print('KB3a4: ',end=''); pprint(KB3a4);
print('SOS3a4: ',end=''); pprint(SOS3a4); print()
prove3(KB3a4,SOS3a4); print()
```

```
test 1
------
KB3a1: [ { B } ]
SOS3a1: [ { -B } ]

[ { -B } ]
Unsatisfiable
1 steps were taken


test 2
------
KB3a2: [ { A }, { B, -A }, { C, -B }, { D, -C } ]
SOS3a2: [ { -D } ]

[ { -D } ]
[ { -C } ]
[ { -B } ]
[ { -A } ]
Unsatisfiable
4 steps were taken


test 3
------
KB3a3: [ { A }, { B, C, -A }, { D, -C }, { E, -C }, { -E } ]
SOS3a3: [ { -B } ]

[ { -B } ]
[ { C, -A } ]
[ { C }, { D, -A }, { E, -A } ]
```

```
[ { D, -A }, { E, -A }, { D }, { E } ]
[ { E, -A }, { D }, { E }, { D } ]
[ { D }, { E }, { D }, { E }, { -A } ]
[ { E }, { D }, { E }, { -A } ]
Unsatisfiable
7 steps were taken


test 4
------
KB3a4: [ { A }, { B, C, -A }, { D, -C }, { E, -C } ]
SOS3a4: [ { -E } ]

[ { -E } ]
[ { -C } ]
[ { B, -A } ]
[ { B } ]
[  ]
Satisfiable
4 steps were taken
```

### 1.4.2 Part B

Now take the same clause set as in Problem Two Part B, but move the last clause into the SOS and keep the other three in KB. Run this example with prove3.

[12]:
```
print('test 3b','\n------')
KB3b1 = [ {-1, 2}, {1, 2}, {-1, -2} ]
SOS3b1 = [ {1, -2} ]

print('KB3b1: ',end=''); pprint(KB3b1);
print('SOS3b1: ',end=''); pprint(SOS3b1); print()
prove3(KB3b1,SOS3b1); print()
```

```
test 3b
------
KB3b1: [ { B, -A }, { A, B }, { -A, -B } ]
SOS3b1: [ { A, -B } ]

[ { A, -B } ]
[ { B, -B }, { A, -A }, { A }, { -B } ]
[ { A, -A }, { A }, { -B }, { B, -A }, { A, B }, { -B, -A }, { -B } ]
[ { A }, { -B }, { B, -A }, { A, B }, { -B, -A }, { -B }, { B, -A }, { A, B }, {
-A, -B }, { A }, { B, -A }, { A, B }, { -A, -B } ]
[ { -B }, { B, -A }, { A, B }, { -B, -A }, { -B }, { B, -A }, { A, B }, { -A, -B
}, { A }, { B, -A }, { A, B }, { -A, -B }, { B }, { -B }, { B }, { -B }, { B },
{ -B }, { B }, { -B } ]
```

```
Unsatisfiable
5 steps were taken
```

### 1.4.3 Part C

Now take the same clause set as in Problem Two Part C, but put the last clause in the SOS and the other 7 in KB. Experiment in the same way you did in Problem Two Part C.

[13]:
```
print('test 3c','\n------')
KB3c1 = [ {1, 2, 3}, {-1,2,3}, {1, 2, -3}, {1, -2, -3}, {1, -2, 3}, {-1, 2,␣
 ↪-3}, {-1, -2, -3} ]
SOS3c1 = [ {-1, -2, 3} ]

print('KB3c1: ',end=''); pprint(KB3c1);
print('SOS3c1: ',end=''); pprint(SOS3c1); print()
prove3(KB3c1,SOS3c1, 14, False); print()

# The number goes to 14 even with the SOS modifications (my kernel died for␣
 ↪larger numbers)
```

```
test 3c
------
KB3c1: [ { A, B, C }, { B, C, -A }, { A, B, -C }, { A, -C, -B }, { A, C, -B }, {
B, -C, -A }, { -C, -A, -B } ]
SOS3c1: [ { C, -A, -B } ]

The empty clause cannot be found after 14 steps.
```

### 1.4.4 Part D

Provide a brief statement of what you observed using the set of support strategy. Explain why you think this behavior is happening.

With the support strategy, the satisfiable clause sets took less time to prove themselves, while the unsatisfiable sets took similar amounts of time; the SOS strategy likely accelerated the proving of satisfiable sets since those satisfiable sets were already accounted for during each step of this algorithm's loop.

## 1.5 Problem 4 (Removing Useless Clauses) (10 pts)

Keeping the set of support strategy, in this problem you must rewrite your prover from Problem 3 as `prove4`, which checks clauses as they are generated by resolution, and does NOT put them in the queue if

- They are tautologies (some symbol and its negation both appear in the clause);
- They have already been generated before.

For the second, you must create a set `Visited` which is a set (hash table) containing copies of all clauses generated at any time in the past; in order to create a set of sets, you will need to make

copies of the clauses as `frozensets`. Here is a brief example of how to do that:

[14]:
```
S = set()
A = {1,2}

S.add( frozenset(A) )
print(S)
print( A in S)
```

```
{frozenset({1, 2})}
True
```

### 1.5.1 Part A

Write `prove4` as specified; you probably want to create a separate function `isTautology(C)`. Test it on examples as in Problem 3, Part A, just to verify you have done it correctly.

[15]:
```
import math
def isTautology(C):
    return any(-1*i in C for i in C)

# tests
print(isTautology({1, 2, -3}))
print(isTautology({}))
print(isTautology({1, -1, 2}))
```

```
False
False
True
```

[16]:
```
def prove4(KB, SOS, limit=30,trace=True):
    queue = SOS.copy()
    for i in range(limit):
        if trace:
            pprint(queue)
        if queue == []:
            print("Satisfiable")
            print(str(i)+" steps were taken")
            return True
        curr = queue.pop(0)
        queue2 = KB.copy() + queue.copy()
        S = set()
        for y in resolveAll(queue2, curr):
            if y == set():
                print("Unsatisfiable")
                print(str(i+1)+" steps were taken")
                return False
            elif isTautology(y) or y in S:
                continue
```

```
            else:
                queue.append(y)
                S.add(frozenset(y))

    print("The empty clause cannot be found after "+str(limit)+" steps.")
    return False


print('\ntest 1','\n------')
KB3a1 = [ {2} ]
SOS3a1 = [ {-2} ]

print('KB3a1: ',end=''); pprint(KB3a1);
print('SOS3a1: ',end=''); pprint(SOS3a1); print()
prove4(KB3a1,SOS3a1); print()

# etc for other 3

print('\ntest 2','\n------')
KB3a2 = [ {1}, {-1,2}, {-2,3}, {-3,4} ]
SOS3a2 = [ {-4} ]

print('KB3a2: ',end=''); pprint(KB3a2);
print('SOS3a2: ',end=''); pprint(SOS3a2); print()
prove4(KB3a2,SOS3a2); print()

print('\ntest 3','\n------')
KB3a3 = [ {1}, {-1,2,3}, {-3,4}, {-3,5}, {-5} ]
SOS3a3 = [ {-2} ]

print('KB3a3: ',end=''); pprint(KB3a3);
print('SOS3a3: ',end=''); pprint(SOS3a3); print()
prove4(KB3a3,SOS3a3); print()

print('\ntest 4','\n------')
KB3a4 = [ {1}, {-1,2,3}, {-3,4}, {-3,5} ]
SOS3a4 = [ {-5} ]

print('KB3a4: ',end=''); pprint(KB3a4);
print('SOS3a4: ',end=''); pprint(SOS3a4); print()
prove4(KB3a4,SOS3a4); print()
```

```
test 1
------
KB3a1: [ { B } ]
SOS3a1: [ { -B } ]
```

```
[ { -B } ]
Unsatisfiable
1 steps were taken


test 2
------
KB3a2: [ { A }, { B, -A }, { C, -B }, { D, -C } ]
SOS3a2: [ { -D } ]

[ { -D } ]
[ { -C } ]
[ { -B } ]
[ { -A } ]
Unsatisfiable
4 steps were taken


test 3
------
KB3a3: [ { A }, { B, C, -A }, { D, -C }, { E, -C }, { -E } ]
SOS3a3: [ { -B } ]

[ { -B } ]
[ { C, -A } ]
[ { C }, { D, -A }, { E, -A } ]
[ { D, -A }, { E, -A }, { D }, { E } ]
[ { E, -A }, { D }, { E }, { D } ]
[ { D }, { E }, { D }, { E }, { -A } ]
[ { E }, { D }, { E }, { -A } ]
Unsatisfiable
7 steps were taken


test 4
------
KB3a4: [ { A }, { B, C, -A }, { D, -C }, { E, -C } ]
SOS3a4: [ { -E } ]

[ { -E } ]
[ { -C } ]
[ { B, -A } ]
[ { B } ]
[  ]
Satisfiable
4 steps were taken
```

### 1.5.2 Part B

Now test it on the example from Problem Three, Part B.

[17]:
```
print('test 4b','\n------')
KB4b1 = [ {-1, 2}, {1, 2}, {-1, -2} ]
SOS4b1 = [ {1, -2} ]

print('KB4b1: ',end=''); pprint(KB4b1);
print('SOS4b1: ',end=''); pprint(SOS4b1); print()
prove4(KB4b1,SOS4b1); print()
```

```
test 4b
------
KB4b1: [ { B, -A }, { A, B }, { -A, -B } ]
SOS4b1: [ { A, -B } ]

[ { A, -B } ]
[ { A }, { -B } ]
[ { -B }, { B }, { -B } ]
Unsatisfiable
3 steps were taken
```

### 1.5.3 Part C

Now test it on the example from Problem Three, Part C, and experiment with it as before to see how far you can search in about one minute.

[18]:
```
print('test 4c','\n------')
KB4c1 = [ {-1, -2, 3}, {-1,2,3}, {-1, -2, -3}, {1, -2, -3}, {1, -2, 3}, {-1, 2,␣
 ↪-3}, {1, 2, -3} ]
SOS4c1 = [ {1, 2, 3} ]

print('KB4c1: ',end=''); pprint(KB4c1);
print('SOS4c1: ',end=''); pprint(SOS4c1); print()
prove4(KB4c1,SOS4c1, 100, False); print()

# Somehow the resolver finishes in 18 steps
```

```
test 4c
------
KB4c1: [ { C, -A, -B }, { B, C, -A }, { -C, -A, -B }, { A, -C, -B }, { A, C, -B
}, { B, -C, -A }, { A, B, -C } ]
SOS4c1: [ { A, B, C } ]

Unsatisfiable
18 steps were taken
```

### 1.5.4 Part D

Summarize what you observed. Does this strategy make a significant difference?

(If you have an interest, you might try this on a similar example with 4 symbols….)

This strategy decreases the time needed to run on several tests; it even cut off 2 steps from part B to prove its unsatisfiability.

## 1.6 Problem Five (Shortest-Clauses-First) (10 pts)

Now you should take your latest `prove4` and add one more refinement: make your queue a priority queue ordered by size, with smaller clauses to the front.

The easiest way to do this (certainly not the most efficient) is to simply sort the queue every time you add something to the end. You can sort a list of sets/lists using as key the size of the member set/list, as illustrated here:

[19]:
```
A = [ {1,2}, {2,3,4}, {1}, {3}, {3,4}]

A.sort(reverse=True, key=(lambda x: len(x)))            # sorts in place

A
```

[19]:
```
[{2, 3, 4}, {1, 2}, {3, 4}, {1}, {3}]
```

### 1.6.1 Part A

Rewrite `prove4` as `prove5` with a priority queue instead of a queue, as suggested, and test it on the same simple examples as in Part A of the previous two problems, to verify that it works as expected.

[20]:
```
def prove5(KB, SOS, limit=30,trace=True):
    queue = SOS.copy()
    for i in range(limit):
        if trace:
            pprint(queue)
        if queue == []:
            print("Satisfiable")
            print(str(i)+" steps were taken")
            return True
        curr = queue.pop(0)
        queue2 = KB.copy() + queue.copy()
        S = set()
        for y in resolveAll(queue2, curr):
            if y == set():
                print("Unsatisfiable")
                print(str(i+1)+" steps were taken")
                return False
            elif isTautology(y) or y in S:
```

```
                continue
            else:
                queue.append(y)
                S.add(frozenset(y))
        queue.sort(reverse=True, key=(lambda x: len(x)))

    print("The empty clause cannot be found after "+str(limit)+" steps.")
    return False


print('\ntest 1','\n------')
KB5a1 = [ {2} ]
SOS5a1 = [ {-2} ]

print('KB5a1: ',end=''); pprint(KB5a1);
print('SOS5a1: ',end=''); pprint(SOS5a1); print()
prove5(KB5a1,SOS5a1); print()

# etc for other 3

print('\ntest 2','\n------')
KB5a2 = [ {1}, {-1,2}, {-2,3}, {-3,4} ]
SOS5a2 = [ {-4} ]

print('KB5a2: ',end=''); pprint(KB5a2);
print('SOS5a2: ',end=''); pprint(SOS5a2); print()
prove4(KB5a2,SOS5a2); print()

print('\ntest 3','\n------')
KB5a3 = [ {1}, {-1,2,3}, {-3,4}, {-3,5}, {-5} ]
SOS5a3 = [ {-2} ]

print('KB5a3: ',end=''); pprint(KB5a3);
print('SOS5a3: ',end=''); pprint(SOS5a3); print()
prove4(KB5a3,SOS5a3); print()

print('\ntest 4','\n------')
KB5a4 = [ {1}, {-1,2,3}, {-3,4}, {-3,5} ]
SOS5a4 = [ {-5} ]

print('KB5a4: ',end=''); pprint(KB5a4);
print('SOS5a4: ',end=''); pprint(SOS5a4); print()
prove4(KB5a4,SOS5a4); print()
```

```
test 1
------
KB5a1: [ { B } ]
```

```
SOS5a1: [ { -B } ]

[ { -B } ]
Unsatisfiable
1 steps were taken


test 2
------
KB5a2: [ { A }, { B, -A }, { C, -B }, { D, -C } ]
SOS5a2: [ { -D } ]

[ { -D } ]
[ { -C } ]
[ { -B } ]
[ { -A } ]
Unsatisfiable
4 steps were taken


test 3
------
KB5a3: [ { A }, { B, C, -A }, { D, -C }, { E, -C }, { -E } ]
SOS5a3: [ { -B } ]

[ { -B } ]
[ { C, -A } ]
[ { C }, { D, -A }, { E, -A } ]
[ { D, -A }, { E, -A }, { D }, { E } ]
[ { E, -A }, { D }, { E }, { D } ]
[ { D }, { E }, { D }, { E }, { -A } ]
[ { E }, { D }, { E }, { -A } ]
Unsatisfiable
7 steps were taken


test 4
------
KB5a4: [ { A }, { B, C, -A }, { D, -C }, { E, -C } ]
SOS5a4: [ { -E } ]

[ { -E } ]
[ { -C } ]
[ { B, -A } ]
[ { B } ]
[  ]
Satisfiable
4 steps were taken
```

### 1.6.2 Part B

Again, test your new prover on the example from Part B of the previous two problems.

[21]:
```
print('test 5b','\n------')
KB5b1 = [ {-1, 2}, {1, 2}, {-1, -2} ]
SOS5b1 = [ {1, -2} ]

print('KB5b1: ',end=''); pprint(KB5b1);
print('SOS5b1: ',end=''); pprint(SOS5b1); print()
prove5(KB5b1,SOS5b1); print()
```

```
test 5b
------
KB5b1: [ { B, -A }, { A, B }, { -A, -B } ]
SOS5b1: [ { A, -B } ]

[ { A, -B } ]
[ { A }, { -B } ]
[ { -B }, { B }, { -B } ]
Unsatisfiable
3 steps were taken
```

### 1.6.3 Part C

Again, test your new prover on the example from Part C of the previous two problems.

[22]:
```
print('test 5c','\n------')
KB5c1 = [ {1, 2, 3}, {-1,2,3}, {1, 2, -3}, {1, -2, -3}, {1, -2, 3}, {-1, 2,␣
 ↪-3}, {-1, -2, -3} ]
SOS5c1 = [ {-1, -2, 3} ]

print('KB5c1: ',end=''); pprint(KB5c1);
print('SOS5c1: ',end=''); pprint(SOS5c1); print()
prove5(KB5c1,SOS5c1, 1000, False); print()

# The prover goes up to 1000+ steps in a minute!
```

```
test 5c
------
KB5c1: [ { A, B, C }, { B, C, -A }, { A, B, -C }, { A, -C, -B }, { A, C, -B }, {
B, -C, -A }, { -C, -A, -B } ]
SOS5c1: [ { C, -A, -B } ]

The empty clause cannot be found after 1000 steps.
```

### 1.6.4 Part D

What did you observe? Did this make a significant difference in the amount of time it took to complete the proof?

Finally, do you think these refinements would reduce the exponential search space for such problems to polynomial size, or is exponential growth still a problem? (Hint: think about what you learned in your algorithms or complexity class about the Satisfiability Problem.)

Response:

The prover was able to go through many more steps in a shorter amount of time, notably demonstrated by Part C.

These refinements do not reduce the exponential search space; satisfiability with more than 2 literals per clause in CNF form is seen as a problem in NP; which means that a solution for it cannot exist in polynomial time if P!=NP.

## 1.7 Problem Six (10 pts)

In this problem, we will use the fastest method, `prove5`, to solve some word problems which can be encoded as problems in propositional logic. For each, give the encoding of the statements in the problem as symbols. (You might want to review pp.32-33 in the textbook before starting.)

### 1.7.1 Part A

Do problem 2.10 from your textbook.

We can convert the sentences into the following statements:

Statement 1: $A \implies C \equiv \neg A \vee C$ Statement 2: $(\neg A \wedge \neg K) \vee (A \wedge K) \equiv (\neg A \vee K) \wedge (A \vee \neg K)$ Statement 3: $K$

Since $K$ is true, we can substitute it into S2: $(\neg A \wedge F) \vee (A \wedge T) \equiv F \vee A \equiv A$, which can then be substituted into S1: $F \vee C \equiv C$; the criminal therefore had a car.

### 1.7.2 Part B

A man and a woman are talking. "I am a man" said the person with black hair. "I am a woman" said the person with white hair. At least one of them is lying. Formalize the puzzle using propositional logic and show using resolution that both of them are lying.

We represent the statement as $(M \wedge \neg W) \vee (\neg M \wedge W)$, where $M$ represents being a man and $W$ represents being a woman. Based on its assignments, only one of $M$ or $W$ can be true at once to make the logical statement true. By resolving the statements, we also obtain an empty clause; this means that there is no satisfying interpretation and both people are lying.

### 1.7.3 Part C

Three boxes are presented to you. One contains gold, the other two are empty. Each box has imprinted on it a clue as to its contents; the clues are:

- Box 1 "The gold is not here"
- Box 2 "The gold is not here"

- Box 3 “The gold is in Box 2”

Only one message is true; the other two are false. Which box has the gold?

Formalize the puzzle using propositional logic and find the solution using resolution.

We can convert the clues into logical statements as follows:

Statement 1: $\neg B_1$ Statement 2: $\neg B_2$ Statement 3: $B_3 = B_2$ Truth: $S1 \vee S2 \vee S3$

If we assign $B_1$ to $T$, then S1 and S3 would both be false; if we assign $B_2$ to $T$, then S2 and S3 both evaluate to false; if we assign $B_3$ to $T$, then S2 and S3 both evaluate to true, and by process of elimination (the other two possibilities are invalid), we know that the third box has the gold.

[23]:
```
# Part A Code
print('Part A','\n------')
KB6a1 = [ {3}, {-1, 2}, {-1, 3} ]
SOS6a1 = [ {1, -3} ]

print('KB6a1: ',end=''); pprint(KB6a1);
print('SOS6a1: ',end=''); pprint(SOS6a1); print()
prove3(KB6a1,SOS6a1, 100, False); print()
```

```
Part A
------
KB6a1: [ { C }, { B, -A }, { C, -A } ]
SOS6a1: [ { A, -C } ]

Satisfiable
70 steps were taken
```

[24]:
```
# Part B Code
print('Part B','\n------')
KB6b1 = [ {1, -2} ]
SOS6b1 = [ {-1, 2} ]

print('KB6b1: ',end=''); pprint(KB6b1);
print('SOS6b1: ',end=''); pprint(SOS6b1); print()
prove5(KB6b1,SOS6b1, 30, False); print()
```

```
Part B
------
KB6b1: [ { A, -B } ]
SOS6b1: [ { B, -A } ]

Satisfiable
1 steps were taken
```

[25]:
```
# Part C Code
print('Part C','\n------')
KB6c1 = [ {-1}, {-2}, {2, -3} ]
SOS6c1 = [ {-2, 3} ]

print('KB6c1: ',end=''); pprint(KB6c1);
print('SOS6c1: ',end=''); pprint(SOS6c1); print()
prove5(KB6c1,SOS6c1, 30, False); print()
```

```
Part C
------
KB6c1: [ { -A }, { -B }, { B, -C } ]
SOS6c1: [ { C, -B } ]

Satisfiable
1 steps were taken
```

### 1.8 Problem Seven (First-Order Logic) (5 pts)

Do problem 3.1 from your textbook. You may provide your solution using ASCII text; by handwriting, scanning, and pasting; or (preferably) in Latex.

Solutions:

a) $\forall x, male(x) \iff \neg(female(x))$

b1) $\forall x, father(x) \iff male(x) \wedge \exists y \exists z, child(y, x, z)$

b2) $\forall x, mother(x) \iff female(x) \wedge \exists y \exists z, child(y, x, z)$

c) $\forall x \forall y, siblings(x, y) \iff \exists a \exists b, child(x, a, b) \wedge child(y, a, b)$

d) $\forall x \forall y \forall z, parents(x, y, z) \iff male(x) \wedge female(y) \wedge child(z, x, y)$

e) $\forall x \forall y, uncle(x, y) \iff male(x) \wedge \exists a, siblings(x, a) \wedge \exists b, child(y, a, b)$

f) $\forall x \forall y, ancestor(x, y) \iff descendant(y, x)$

Note that $descendant(x, y)$ is defined in the book as follows: $\forall x \forall y, descendant(x, y) \iff \exists z, child(x, y, z) \vee (\exists u \exists v, child(x, u, v) \wedge descendant(u, y))$.

### 1.9 Problem Eight (First-Order Logic) (5 pts)

Do problem 3.2 from your textbook. You may provide your solution using ASCII text; by handwriting, scanning, and pasting; or (preferably) in Latex.

Solutions:

a) $\forall x, person(x) \implies \exists y \exists z mother(y) \wedge father(z) \wedge parents(y, z, x)$

b) $\exists x, person(x) \wedge father(x) \vee mother(x)$

c) $\forall x, isbird(x) \implies canfly(x)$

d) $\exists x \exists y, animal(x) \land animal(y) \land eatsgrains(y) \land caneat(y, x)$, caneat is defined as "y can be consumed by x"

e) $\forall x, animal(x) \land (\exists y, plant(y) \land caneat(y, x)) \lor (\exists z, animal(z) \land caneat(z, x) \land smallerthan(z, x))$

### 1.10 Problem Nine (First-Order Logic) (5 pts)

Do problem 3.4 from your textbook. You may provide your solution using ASCII text; by handwriting, scanning, and pasting; or (preferably) in Latex.

Rule 1: $\forall x \forall y, lessthan(x, y) \lor equals(x, y) \lor greaterthan(x, y)$

Rule 2: $\forall x \forall y, lessthan(x, y) \iff greaterthan(y, x)$

Rule 3: $\forall x \forall y \forall z, lessthan(x, y) \land lessthan(y, z) \implies lessthan(x, z)$

### 1.11 Problem Ten (First-Order Logic) (5 pts)

Do problem 3.5 from your textbook. You may provide your solution using ASCII text; by handwriting, scanning, and pasting; or (preferably) in Latex.

a) MGU: $x \to f(z), y \to z, u \to f(y)$; $p(x, f(y)), p(f(z), u) \equiv p(f(z), f(y)), p(f(z), f(y))$

b) The two expressions cannot be unified; f(x) cannot be unified with y.

c) The two expressions cannot be unified; there is no assignment that allows the statements to be equivalent to each other.

d) MGU: $x \to 3$; $3 < 2 * 3 \equiv 3 < 6$

e) MGU: $x \to g(w, w), y \to g(x, x), z \to g(y, y), u \to f(g(w, w), g(x, x), g(y, y))$; $q(f(x, y, z), f(g(w, w), g(x, x), g(y, y))), q(u, u) \equiv f(g(w, w), g(x, x), g(y, y))$ $f(g(w, w), g(x, x), g(y, y))$