SAT SOLVER

ROLL NO: 160101052
NAME: POREDDY SAIKIRAN REDDY

Implemented an SAT solver using CDCL method using non-chronological Backtracking.
The code contains the following sections:

1. Parsing the Input
2. Output
3. CDCL Algorithm
    a. Preprocessing
    b. Decide
    c. Conflict Detection
    d. Conflict Analysis
    e. Resolution
    f. Backtrack

Parsing Input:

The input file is parsed gathering formula in terms of a list of list (denoted by eq). Each object in the list is a clause which is also a list. A clause contains integers as parsed from the formula representing a literal defined by a variable. Negative values represent the negation of a particular variable. Novar contains the number of variables.
Eg: [[1,2,-3],[-1,2]] represents formula $(x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2)$

```
#parse the file for clauses
def file_parse(finput):
    lines = open(finput,'r').readlines()
    eq = []
    firstline = lines[0].split()

    #number of variables is stored
    novar = int(firstline[2])

    #append clauses as lists
    for line in lines[1:]:
        temp = []
        temp = [int(x) for x in line.split()]
        temp.pop()
        eq.append(temp)
```

Output:

The output format is Number of Clauses, Number of Variables, Status of Assignment, Number of Learned Clauses, Number of Decisions and Number of Implications. Restart was not implementable in this idea.

```
#Apply the CDCL algorithm and print the output
output,lc_count,des_count,imp_count = CDCL(eq,novar)
print(f'Number of Variables: {novar}')
print(f'Number of Clauses: {len(eq) - lc_count}')
print(f'Assignment: {output}')
print(f'Learned Clauses: {lc_count}')
print(f'Descisions: {des_count}')
print(f'Implications: {imp_count}')

#end the timer and calculate the runtime
end = time.time()
print(f'Runtime: {end - start}s')
```

CDCL Algorithm:

The algorithm used is Conflict driven Clause Learning, which upgrades over DPLL by applying conflict analysis and using non-chronological backtracking. We remove unnecessary branching on the variables by finding the variables which cause the conflict and learn a new clause which doesn't let the same conflict happen again. This improves the algorithm manifold over DPLL. The following stages were used in CDCL algorithm.

Notations:

v : denotes list where we'll be storing the variables, assigned value and decision level.

anc_cl : denotes the index of antecedents of the given implied literal in eq. For decision and unary clause literals, it's stored as -1.

var : is a list which contains all the decided variables and implied variables, it ensures termination of the program when all the variables are assigned.

confl_Detect: Function to detect conflict

confl_anal: Function to analyze conflict and learn a new clause

resolve: Function for resolution
backtrack: Function to implement backtracking
check: check if the given literal is in v or no append it.
update_var: number of variables assigned is stored
lit_freq: List which stores literal frequency
orig_lit_freq: List which stores the original literal frequency

Algorithm:

```
CDCL(eq):
       v = []
       anc_cl = {}
       preprocessing()
       while ( an unassigned variable exists):
              dl = dl +1
              decide()
              while( confl_Detect(eq,v,dl) detects conflict):
                b, c = confl_anal(eq,v,dl)
                eq = eq ∪ {c}
                if(b < 0):
                    return "UNSAT"
                v  = backtrack(eq,v,b)
       return "SAT"
```

Preprocessing:

In this, we will be adding all the unary clauses to the variable storing list (v) and remove all the clauses which contain the same literal and assignment, but not the unary clause(As It will help to find conflict later). It'll help in reducing the size of formula (eq). Literal frequency is found and stored both negated and non-negated are taken as 1 frequency. Polarity determines how many negated and how many non-negated instances of a literal are present. If polarisation of a literal is negative then it means negated literal is more common and vice versa. Also, store the original values in case we change the current literal frequency. This will help as the size of maintaining and processing a copy which we use in Conflict Detection. We also run for finding conflict as the decision level is 0. Which implies conflict at this stage leads to UNSATISFIABILITY.

```python
    #If the equation contains unary clauses add them to v
    for item in eq:
        if len(item) == 1:
            if item[0] < 0:
                x = -1*int(item[0])
                v.append([x,0,0])
                anc_cl[-1*x] = -1
            else:
                x = int(item[0])
                v.append([x,1,0])
                anc_cl[x] = -1

    #Preprocessing deletion of clauses which will be always be satisfiable
    #except the unary clause for 0 dl vars
    eq = pre_del(eq,v)
    if eq == []:
        return 'SAT',0,0,imp_count

    #initalize literal frequency and literal polarizarion
    lit_freq= [0 for x in range(1,novar+1)]
    lit_pol=[0 for x in range(1,novar+1)]

    #find the values of literal frequency
    # and literal polarization
    for clause in eq:
        for literal in clause:
            if literal > 0:
                lit_freq[literal-1] = lit_freq[literal-1] + 1
                lit_pol[literal-1] = lit_pol[literal-1] + 1

            if literal < 0:
                lit_freq[-1*literal-1] = lit_freq[-1*literal-1] + 1
                lit_pol[-1*literal-1] = lit_pol[-1*literal-1] - 1

    #store the original values in another list
    orig_lit_freq = copy.deepcopy(lit_freq)
```

Decide:

We decide what variable to brach next with an assumption, for this the implemented part just contains finding one after another value until it finds the variable unassigned. Increment the decision level by 1.

```python
            #increment descision level and decide next variable
            dl = dl+1
            for i in range(1,novar+1):
                L = []
                for item in v:
                    L.append(item[0])
                if i not in L:
                    v.append([i,1,dl])
                    anc_cl[i] = -1

                    # increment descisions counter
                    des_count = des_count + 1
                    break
```

Decide:

Decide is used to find the next variable which is not present in the list ‘v’. This is found from the list maintained consisting of frequencies. The index value with maximum frequency is taken as the next variable. The sign i.e negated or not is decided from the polarity of the literal if polarity negative then negative value of literal is added to ‘v’ and vice-versa. This is because a large number of equations becomes satisfiable upon that decision.

```
#function to find new variable to assign
def decide(var,novar,lit_freq,lit_pol,orig_lit_freq,confl_count):

    #assign frequency for variables in var
    for i in var:
        if lit_freq[i-1] != -1:
            lit_freq[i-1] = -1

    #decay the frequency as the
    #number of conflicts reach 100
    if confl_count > 100:
        confl_count = confl_count%100
        for i in range(0,novar):
            orig_lit_freq[i] = orig_lit_freq[i]/2
            if lit_freq[i] != -1 :
                lit_freq[i] = lit_freq[i]/2

    #find the index with maximum frequency
    var = lit_freq.index(max(lit_freq))

    #assign sign based on polarisation value
    if lit_pol[var] > 0:
        sign = 1
    else:
        sign = 0

    return var+1,sign,lit_freq,orig_lit_freq
```

Conflict Detection:

In conflict Detection for each variable, we apply unit propagation and find implied literals and set the antecedent for the implied variables, if we find an empty list it implies we have found a conflict. The reason for using deep copy and preparing a copy for the formula is not to alter the contents of the formula in case we need to backtrack. If a conflict is found set the antecedent value.

```python
def confl_Detect(eq,v,anc_cl,dl,imp_count):

    #copy the contents of eq
    eq_copy = []
    eq_copy = copy.deepcopy(eq)

    for i in v:
        # z represents the negated value of the variable in v
        z = int(i[0])
        if i[1] == 1:
            z = -1*z
        # index is used to find the index of clause which is ancedent for a new variable which will be added
        index = -1
        for lis in eq_copy:
            index = index + 1
            #if -z exists means satisfiable clauses
            if -1*z in lis:
                lis = []
                lis.append(-1*z)
            #if the value exists remove from lis
            if z in lis:
                lis.remove(z)
            #if a variable is implied add to v and set its antecedent clause
            if len(lis) == 1:
                x = lis[0]
                y,v = check(x,v,dl)
                if y == 1 and lis[0] != -1*z:
                    imp_count = imp_count + 1
                    anc_cl[x] = index
            #if a conflict is detected
            if len(lis) == 0:
                anc_cl[0] = index
                return 0,v,anc_cl,imp_count

    return 1,v,anc_cl,imp_count
```

Conflict Analysis:

For conflict analysis, we first need to find the level of conflict. From the decision variable in order to reach the conflict, we can think of a path constructed from the implied variables. The point through which we definitely need to pass is called Unique Implication Point. We can take note that the decision variable is also a UIP. We start at the conflict for which we know the antecedent from which we can construct the corresponding clause, for each of the variables in the antecedent at the same decision level we find the antecedent. For these two antecedents, we can apply binary resolution rule and find a clause without the selected variable from the first antecedent. We take this as the new list and find a new variable at the same decision level to repeat the process again until we reach the UIP (i.e decision variable). The new clause will contain only the decision variable at this level and literals from previous decisions. This will be added to the formula as learned clause and this prevents the condition for conflict.

We return two things, one is the asserting level for which we need to backtrack and another the new learnt clause. The asserting level determines the level at which the decision before the current decision takes place in the learnt clause. The deletion was not used if there is a unary learned clause because removal from original equation would lead to storing inaccurate indexes in anc_lc{}

```python
# store number of literals found from same descision level
count = 0
while(1):
    count = 0

    #iterate over all the literals to find UIP
    for i in lis:
        for j in v:
            if j[0] == abs(i) and j[2] == conf_l:
                count = count+1
        #ancedent of resol will be used to resolve and find UIP
        for j in v:
            if j[0] == abs(i) and j[2] == conf_l and anc_cl[-1*i] != -1:
                resol = i

    # implies there is only the descision variable and its the UIP
    if count == 1:
        break

    #store the antecedent and resolve
    ante = eq[anc_cl[-1*resol]]
    lis = resolve(ante,lis,resol)
```

Resolution:

We apply resolution rule to remove an unwanted variable from a set of two clauses in one we need to have the literal and in another a negated literal, we remove the literals and join them by conjunction. ex: $x_1 \lor x_2 \lor \neg x_3$ and $\neg x_4 \lor x_5 \lor x_3$ can be joined as $x_1 \lor x_2 \lor \neg x_4 \lor x_3$ as at least one of x_3 , $\neg x_3$ must be true. for either of the assignment we the resolved clause must be true.

```python
#function to perform resolve
def resolve(ante,lis,x):

    #copy the two lists into new lists
    new_lis = copy.deepcopy(ante)
    temp = copy.deepcopy(lis)

    #create new list and apply binary resolution rule
    merge = []

    for i in new_lis:
        merge.append(i)
    for i in temp:
        merge.append(i)
    merge.remove(x)
    merge.remove(-1*x)

    # remove duplicates from the list
    res = []
    for i in merge:
        if i not in res:
            res.append(i)

    return res

```

Backtrack:

We backtrack to the level we get from conflict analysis by undoing all the antecedents, decision assigned variable and implied variables.

```python
#function to implement backtrack
def backtrack(v,b,anc_cl):

    # create a list to store all the
    # variables which needs to be removed
    rem = []

    #iterate over v to find all the variables
    #with dl greater than b and also remove antecedent
    for i in v:
        x = i[0]
        if i[1] == 0:
            x = -1*x
        if i[2] >= b:
            rem.append(i)
            if x in anc_cl.keys():
                del anc_cl[x]

    #remove literals
    for j in rem:
        v.remove(j)

    #remove the antecedent of the conflict
    del anc_cl[0]

    return v,anc_cl,b
```