Instituto Superior Técnico

IASD

Inteligência Artificial e Sistemas de Decisão

Laboratório Nr. 2

Autores:
Eduardo Lima Simões da Silva, 69916
José Pedro Braga de Carvalho Vieira Pereira, 70369

21 de Novembro de 2014

Contents

1 Introdução

Este relatório apresenta a resolução do problema proposto como o segundo laboratório de IASD, no qual se pretende criar um "CNF converter". Desta forma, o objectivo deste laboratório é criar um programa que converte frases lógicas, no formato de lógica proposicional, para o formato CNF (Clausal normal form), sob um conjuncto de disjunções.

O programa tem de ser capaz de receber comandos do utilizador, e consoante o desejado, mostrar e listar as frases lógicas antes e após a conversão, assim como todos os passos intermédios necessários para a conversão. As frases lógicas são recebidas por via de um ficheiro de dados, sendo posteriormente toda a informação pedida pela utilizador gravada noutro ficheiro de dados.

2 Algoritmo

Por forma a uma mais fácil compreensão do algoritmo usado, assim como garantir uma maior flexibilidade e adaptabilidade para a necessidade de usar o mesmo para outros fins, foram usados dois ficheiros de código, separando a função mãe (main.py) que contém o menu e invoca as funções necessárias, da função com as funções e algoritmos de conversão (fu.py) que poderá ser usada para qualquer problema onde seja necessário uma frase lógica na forma CNF.

A primeira acção executada pelo algoritmo é a leitura do ficheiro que contém as frases lógicas, onde este tem de receber e armazenar de uma forma correcta e eficiente toda a informação para posterior conversão. Sendo de salientar que a maneira como esta acção é feita é um elemento chave para a execução do restante algoritmo.

Assim, inicialmente a informação proveniente do ficheiro é recolhida e armazenada numa lista onde cada elemento corresponde a uma frase lógica, e posteriormente, através de um ciclo, cada elemento dessa lista vai ser subdividido de forma recursiva, novamente em forma de lista, assumindo assim a forma de uma 'syntactic tree', onde, o elemento de menor dimensão da lista será a representação de um único símbolo ('atom', 'not', 'and', 'or', '=>', '<=>').

Após as diversas frases lógicas estarem presentes na lista, e consoante o pedido do utilizador, o processo de conversão para o formato CNF inicia-se. Este processo consiste numa sequência de cinco passos que foram implementados através de funções individuais ('step by step') onde se fez recurso de uma implementação recursiva por forma a garantir uma completa e correcta conversão. Adicionalmente, foram criadas diversas funções auxiliares para verificar qual o tipo da clausula a analisar, estas funções retornam 1 caso se verifique que é de facto um determinado tipo lógico.

Sendo agora apresentados os cinco principais passos presentes no algoritmo:

1. Eliminar equivalências($\Leftrightarrow$), onde $(A \Leftrightarrow B)$ fica $(A \Rightarrow B) \wedge (B \Rightarrow A)$. Com o objectivo de executar esta acção a função `solve_equivalence` foi implementada, nela uma frase (lista de entrada) é processada. Se o primeiro elemento da lista de entrada é um equivalente ($\Leftrightarrow$) a função substituí-a por uma nova cláusula cujo primeiro elemento é uma conjunção (and), das duas implicações($\Rightarrow$) tal como mostrado previamente. Sendo esta recursiva, é importante salientar que os elementos escritos como argumentos da implicação (A,B) são também processados pela função em si (processo recusivo), e só posteriormente a função retorna uma lista com a conjunção das implicações geradas. Se o primeiro elemento for outra operação lógica, a função é chamada novamente, mantendo o primeiro argumento de entrada, procura implicações nos respectivos elementos sucessivamente até percorrer toda a operação lógica, pois, em qualquer ponto desta pode haver uma equivalência;

2. Eliminar implicações($\Rightarrow$), onde $(A \Rightarrow B)$ fica $(\neg A \vee B)$. Para resolver esta situação foi implementada outra função com o nome de `solve_implication`, sendo bastante semelhante à mencionada anteriormente. Novamente através dum processo recursivo, a função procura uma implicação em qualquer ponto da frase lógica, e ao encontrar, a função retorna a disjunção da negação do primeiro argumento com o segundo argumento. Contudo, tal como explicado no passo anterior, antes de retornar a disjunção, a função é novamente chamada para cada um dos argumentos, por forma a chegar a todas as implicações que podem estar incluídas na frase lógica (e.g. uma equivalência gerará uma conjunção de duas implicações, sem analisarmos essa conjunção recursivamente não conseguiriamos eliminar essas duas implicações em questão).;

3. Colocar negações($\neg$) para o interior, onde $\neg(A \vee B)$ fica $(\neg A \wedge \neg B)$. À semelhança dos passos anteriores, esta função, `solve_negation`, procura recursivamente negações na frase lógica, ao encontrar, tem dois casos possíveis a analisar, se o argumento da negação for outra negação elimina ambas (dupla negação) e invoca novamente a função para o conteúdo restante, se o argumento for uma conjunção ou disjunção, inverte-a para disjunção ou conjunção respectivamente, negando os dois argumentos da mesma e invoca novamente a função antes de retornar a nova frase lógica.

4. Aplicar a regra distributiva, onde para os casos mais simples $(A \wedge B) \vee E$ fica $(A \vee E) \wedge (B \vee E)$ mas facilmente casos mais complexos são gerados como $(A \wedge E) \vee (B \wedge D)$ que fica $(A \vee B) \wedge (A \vee D) \wedge (E \vee B) \wedge (E \vee D)$. Este facto, juntamente com o facto de simultaneamente termos começado o processo de simplificação, fez com que a função `solve_disjunction` tenha sido especialmente desafiante de desenvolver. Tal como os outros passos, procura-se recursivamente por uma disjunção na frase, ao encontrar, analisamos se há uma conjunção num dos elementos, ou em ambos, aplicando assim a distributiva da forma adequada, como mostram os exemplos acima referidos. Com a distributiva, voltamos a invocar a função para cada um dos restantes argumentos, e confirmamos ainda que os dois argumentos da conjunção são diferentes, pois, se forem iguais, sabendo que $(A \wedge A)$ fica A, removemos a conjunção deixando apenas um dos argumentos.

5. Pôr CNF como um conjunto de disjunções (cláusulas), como exemplo $(\neg A \vee E) \wedge (B \vee E)$ fica $[[[not, A], E], [B, E]]$. Para tal, os argumentos das diversas disjunções são separados de forma recursiva, limitando o tamanho da lista que vai conter as diversas consjunções. As conjunções são então simplificadas para ficarem só os atomos das mesmas presentes na lista, cujo tamanho foi definido pelo numero de disjunções inicial. Durante a execução deste último passo da conversão é também realizada a simplificação dos seguintes casos, possivelmente reduzindo o numero resultante de clausulas:

   - Simplificar conjunções com os mesmo átomo, $((A \vee A \vee B) \wedge (C))$ simplifica-se de modo a ficar$(A \vee B) \wedge (C)$;
   - Eliminar conjunções com os mesmos átomos , como $(A \vee B) \wedge (B \vee A)$ pode ser simplesmente $(A \vee B)$;
   - Simplificar disjunções onde $(A \wedge (A \vee B))$ pode ser simplesmente (A).

Após estes cinco passos foi conseguida a conversão do conjunto de frases lógicas num conjunto de cláusulas. Sendo ainda necessário eliminar possíveis repetições e executar simplificações directas, que não foram tidas em conta nos restantes passos, como cláusulas impossíveis ou mesmo unitárias (e.g. $\neg A \vee A$ é condição unitária). Desta forma, realizou-se um passo extra com vista à final simplificação. Este passo foi implementado através de uma procura, novamente de forma recursiva, ao longo da lista na forma CNF procurando simplificar a frase lógica até ao seu formato irredutivel, simplificando, os seguintes casos:

- Avisar quanto à existência de cláusulas impossíveis, $(A) \wedge (\neg A)$ é uma condição impossível, o que torna a frase lógica impossível (e.g. $(A) \wedge (\neg A) \wedge (B)$, fica $Impossvel \wedge (B)$, que é Impossível;
- Simplificar condições unitárias, como $(A \vee \neg A)$ pode ser simplesmente uma condição unitária no caso de ser a unica conjunção ou pode desaparecer no caso de estar numa disjunção (e.g $(1) \wedge (B)$ é B) ;

3 Resultados

Com este algoritmo, foi obtido assim um conversor de frases lógicas para o formato CNF, sob a forma de conjunto de disjunções, onde a frase lógica é ainda simplificada e reduzida à sua forma mais concisa, em anexo são apresentados vários exemplos de conversões efectuadas para verificar o correcto funcionamento do algoritmo conversor , sendo uma delas abordada 'step-by-step', para uma análise e verificação mais clara.

4 Conclusão

Após a implementação e verificação do algoritmo é possível concluir que o laboratório foi concluído com sucesso. Todas as funções desejáveis e todos os requisitos foram implementados. O conversor de frases lógicas para a forma CNF criado funciona correctamente, tendo apresentado algumas dificuldades para a sua implementação nomeadamente no que diz respeito ao funcionamento recursivo e implementação do quarto passo (regra distributiva), contudo essas dificuldades foram superadas.

References

[1] Luís Custódio, Rodrigo Ventura, North tower – ISR / IST, 2014, *Lecture notes - Course Artificial Intelligence and Decision Systems*

[2] Stuart Russell, Peter Norvig , 2003, Prentice Hall, Second Edition, *Artificial Intelligence: A Modern Approach*

5 Anexos

Entre diversas outras frases lógicas, foram testados os seguintes exemplos, com a respectiva conversão:

1. ('=>', 'Mythical', 'Immortal');
   $[[[' not',' Mythical'],' Immortal']]$
2. ('=>', ('not', 'Mythical'), ('and', ('not', 'Immortal'), 'Mammal'));
   $[[' Mythical', [' not',' Immortal']], [' Mythical',' Mammal']]$
3. ('=>', ('or', 'Immortal', 'Mammal'), 'Horned');
   $[[[' not',' Immortal'],' Horned'], [[' not',' Mammal'],' Horned']]$
4. ('=>', 'Horned', 'Magical');
   $[[[' not',' Horned'],' Magical']]$
5. ('not', ('or', 'A', 'B'));
   $[[[' not',' A']], [[' not',' B']]]$
6. ('=>', ('and', 'A', 'B'), 'C');
   $[[[' not',' A'], [' not',' B'],' C']]$
7. ('<=>', 'A', 'B');
   $[[[' not',' A'],' B'], [[' not',' B'],' A']]$
8. ('=>', 'A', 'A');
   $Unitary, \quad always \quad satisfiable : [[' not',' A'],' A']$
9. ('or', ('=>', 'A', 'A'), ('<=>', 'A', 'A'));
   $Unitary, \quad always \quad satisfiable : [[' not',' A'],' A']$
10. ('<=>', ('not', 'A'), 'B');
    $[[' A',' B'], [[' not',' B'], [' not',' A']]]$
11. ('<=>', 'A', ('not', 'A'));
    $Impossible, \quad cannot \quad satisfy \quad both : [[' not',' A']] and [' A']$
12. ('not', ('<=>', 'A', ('not', 'A')));
    $Unitary, \quad always \quad satisfiable : [' A', [' not',' A']]$
13. ('and', ('not', ('or', 'A', 'B')), ('<=>', ('not', 'A'), 'B'));
    $[[[' not',' A']], [[' not',' B']], [' A',' B'], [[' not',' B'], [' not',' A']]]$
14. ('<=>', ('not', 'A'), ('not', 'A'));
    $Unitary, \quad always \quad satisfiable : [' A', [' not',' A']]$
15. ('<=>', ('not', 'B'), ('not', ('or', 'PT', 'NL')));
    $[[' B', [' not',' PT']], [' B', [' not',' NL']], [' PT',' NL', [' not',' B']]]$
16. ('<=>', ('or', 'PT', 'NL'), ('not', 'EN'));
    $[[[' not',' PT'], [' not',' EN']], [[' not',' NL'], [' not',' EN']], [' EN',' PT',' NL']]$
17. ('<=>', ('and', 'PT', 'NL'), ('and', ('or', 'EN', 'PT'), 'BR'));
    $[[[' not',' PT'], [' not',' NL'],' BR'], [[' not',' EN'], [' not',' BR'],' PT'],$
    $[[' not',' EN'], [' not',' BR'],' NL'], [[' not',' PT'], [' not',' BR'],' NL']]$

18. ('<=>', ('or', ('and', 'PT', ('not', 'NL')), ('and', 'COM', 'EN')), ('and', ('or', ('not', 'DE'), 'BR'), 'BR'));
[[['not',' PT'],' NL',['not',' DE'],' BR'],[['not',' PT'],' NL',' BR'],
[['not',' COM'],['not',' EN'],['not',' DE'],' BR'],[['not',' COM'],['not',' EN'],' BR'],
['DE',['not',' BR'],' PT',' COM'],['DE',['not',' BR'],' PT',' EN'],
['DE',['not',' BR'],['not',' NL'],' COM'],['DE',['not',' BR'],['not',' NL'],' EN'],
[['not',' BR'],' PT',' COM'],[['not',' BR'],' PT',' EN'],[['not',' BR'],['not',' NL'],' COM'],
[['not',' BR'],['not',' NL'],' EN']]

19. ('<=>', ('or', ('and', 'PT', 'NL'), ('and', 'NL', 'PT')), ('and', ('or', 'PT', ('not', 'NL')), 'BR'));
[[['not',' PT'],['not',' NL'],' BR'],[['not',' NL'],['not',' PT'],' BR'],
[['not',' PT'],['not',' BR'],' NL'],['NL',['not',' BR'],' PT'],['NL',['not',' BR'],' PT'],
['NL',['not',' BR']],['NL',['not',' BR'],' PT']]

20. ('<=>', ('or', ('and', 'PT', 'NL'), ('and', 'NL', 'PT')), ('and', ('or', 'PT', ('<=>', 'A', ('not', 'B'))), 'US'));
[[['not',' PT'],['not',' NL'],' US'],[['not',' NL'],['not',' PT'],' US'],
[['not',' PT'],['not',' US'],' NL'],['A',['not',' B'],['not',' US'],' PT',' NL'],
['A',['not',' B'],['not',' US'],' PT'],['B',['not',' A'],['not',' US'],' PT',' NL'],
['B',['not',' A'],['not',' US'],' PT'],['A',['not',' B'],['not',' US'],' NL'],
['A',['not',' B'],['not',' US'],' NL',' PT'],['B',['not',' A'],['not',' US'],' NL'],
['B',['not',' A'],['not',' US'],' NL',' PT']]

Por forma a tornar mais perceptível a forma como é efectuada a conversão, e os respectivos passos efectuados, passa a ser explificado o nono exemplo 'step-by-step':

- Step1, get rid of equivalences (<=>):
$['or',['=>',' A',' A'],['and',['=>',' A',' A'],['=>',' A',' A']]]$
- Step2, get rid of implications (=>):
$['or',['or',['not',' A'],' A'],['and',['or',['not',' A'],' A'],['or',['not',' A'],' A']]]$
- Step3, move negations inwards (not):
$Nr1 : ['or',['or',['not',' A'],' A'],['and',['or',['not',' A'],' A'],['or',['not',' A'],' A']]]$
- Step4, apply the distributive to disjunctions (or):
$['or',['or',['not',' A'],' A'],['or',['not',' A'],' A']]$
- Step5, CNF as a set of disjunctions:
$[[['not',' A'],' A']]$
- Step6, Simplifies clause:
$Unitary, \quad always \quad satisfiable : [['not',' A'],' A']$

6 Código Python

Listing 1: File lab2_main_final.py

```
# Eduardo Silva nr 69916 - MEAer
# Jose Pereira nr 70369 - MEAer
# Version: Python 3.4.1

import copy #To copy lists without using pointers
from time import time #To count computational time
import fu
import os

## MAIN FUNCTION
#Description: executes the convertion between a logical sentence in the format:('=>', 'Mythical',
    'Immortal'),from a file, to the CNF format.
#Reads from input_file.txt, and writes output to comant line and output.txt.
#Inputs: NONE
#Outputs: NONE
read_file = fu.open_file('input_file.txt') #Reads info given in input file, by line
```

```
sentences=fu.file_to_list(read_file) #Convert lines of the file into a list
f = open('output_file.txt', 'w') #Initiate Damp File, that will store everything

flag=1
while flag==1: #MENU : CHOOSE AN OPTION
    print('Artificial Intelligence and Decision Systems (IASD)')
    print('Assignment #2 - Version: Python 3.4.1')
    print('Eduardo Silva nr 69916 - MEAer')
    print('Jose Pereira nr 70369 - MEAer')
    print('\nOptions:\n\t1: List sentences before conversion')
    print('\t2: CNF converter - List One Sentence Step-by-Step')
    print('\t3: CNF converter - List All Sentences\n\t4: Exit')
    opt1=input('Choose an option [1, 2, 3 or 4]\n')
    initial_time = time() #Set initial time

    if opt1=='1': #If chooses '1', print sentences before conversion
        fu.print_steps('List sentences before conversion:', sentences, f) #Print Sentences

    elif opt1=='2': #If chooses '2', asks which sentence to convert to CNF form
        print('Which sentence, from (input_file.txt), do you want to convert?')
        f.write('\nWhich sentence, from (input_file.txt), do you want to convert?\n')
        fu.print_sentences(sentences, f)
        opt2=0
        while (int(opt2) < 1 or int(opt2) > len(sentences)):
              opt2=input('Choose the number of the sentence, from (input_file.txt), to convert\n')
              if (int(opt2) < 1 or int(opt2) > len(sentences)):
                  print("That sentence doesn't exist, please choose again")
        f.write('\nSentence Chosen:'+opt2+'\n')
        initial_time = time() #Reset initial time, needed to wait for user's response
        step1, step2, step3, step4, step5, step6=fu.CNF_converter([sentences[int(opt2)-1]])
            #Conversion Step by Step
        fu.print_steps('Step1, get rid of equivalences (<=>):', step1, f) #Print Step1
        fu.print_steps('Step2, get rid of implications (=>):', step2, f) #Print Step2
        fu.print_steps('Step3, move negations inwards (not):', step3, f) #Print Step3
        fu.print_steps('Step4, apply the distributive to disjunctions (or):', step4, f) #Print Step4
        fu.print_steps('Step5, CNF as a set of disjunctions:', step5, f) #Print Step5
        fu.print_steps('Step6, Simplifies clause:', step6, f) #Print Step6

    elif opt1=='3': #If chooses '3', converts every sentence to CNF form
        step1, step2, step3, step4, step5, step6=fu.CNF_converter(sentences) #Conversion done Step
            by Step as well
        fu.print_steps('\nConverted Sentences to the CNF form:', step6, f) #Print Converted
            Sentences

    elif opt1=='4': #If chooses '4', closes Damp File and Exits
        flag=0
        f.close() #Close Damp File

    else: #If neither of the four options chosen, user asked to choose again
        print('Unknown option, please choose again')

    computational_time=time()-initial_time #Computational Time
    print('\nComputational Time: '+str(computational_time)+' seconds') #Prints Computational Time
         after each option
    print('------------------------------------------------------------')
    if flag==1: #If file still open (doesn't make sense to compute time to exit [0 seconds])
        f.write('\nComputational Time: '+str(computational_time)+' seconds')
        f.write('\n------------------------------------------------------------\n')
```

Listing 2: File ful.py

```
# Eduardo Silva nr 69916 - MEAer
# Jose Pereira nr 70369 - MEAer
# Version: Python 3.4.1

import copy #To copy lists without using pointers
```

```
from time import time #To count computational time

#SET OF MICRO FUNCTIONS TO CHECK IF A GIVEN SENTENCE IS AN ATOM, NEGATION...
#Description:Verifies if a given sentence is an atom, negation, implication, etc
#Inputs: Clause
#Outputs: 1 if is the logical operand wanted, 0 if not

def isequi(sent):
    if sent[0]=='<=>': return 1
    return 0
def isimp(sent):
    if sent[0]=='=>': return 1
    return 0
def isneg(sent):
    if sent[0]=='not': return 1
    return 0
def isor(sent):
    if sent[0]=='or': return 1
    return 0
def isat(sent):
    if sent[0]!= '<=>' and sent[0]!= '=>' and sent[0]!= 'or' and sent[0]!= 'and' and sent[0]!=
        'not':return 1
    return 0
def isand(sent):
    if sent[0]=='and': return 1
    return 0

# OPEN FUNCTION: Reads the file and stores it line by line
#Description:Opens a file and reads each line of its content.
#Inputs: file (name of the file)
#Outputs: read_file (what is in the file)
def open_file(file):
    with open(file) as f:
        read_file=[f.readline()]
        read_file[0]=read_file[0].strip('\n')
        aux=1
        while (aux != ''):
            aux=f.readline()
            aux=aux.strip('\n')
            read_file=read_file+[aux]
    return read_file

# CONVERT FILE TO A LIST: PART B
def listit(t):
    return [listit(i) for i in t] if isinstance(t, (list,tuple)) else t

# CONVERT FILE TO A LIST: PART A
def file_to_list(read_file):
    lista=[]
    for i in range(len(read_file)-1):
        tupline=eval(read_file[i])
        lista.append(tupline)
    nlist=listit(lista)
    return nlist

# SOLVE EQUIVALENCE, get rid of equivalences (<=>)
#Description:Modifies a clause to stay without any equivalence operand according to the rules.
#Inputs: Clause
#Outputs: Clause without any equivalence operand.
def solve_equivalence(clause):
    if isat(clause): # Atom
        return clause # There are no (<=>), done, return itself
    if isimp(clause) or isand(clause) or isor(clause): # =>, or, and
        clause=[clause[0]]+[solve_equivalence(clause[1])]+[solve_equivalence(clause[2])] # Keep the
            symbol, recursively check the rest of the sentence
        return clause
```

```
    if isneg(clause): # Not
        clause=[clause[0]]+[solve_equivalence(clause[1])] # Keep the 'not', recursively check the
            rest of the sentence
        return clause
    if isequi(clause): # Get rid of the equivalence, recursively check the rest of the sentence
        clause=['and', ['=>']+[solve_equivalence(clause[1])]+[solve_equivalence(clause[2])],
            ['=>']+[solve_equivalence(clause[2])]+[solve_equivalence(clause[1])]]
        return clause

# SOLVE IMPLICATION
#Description:Modifies a clause to stay without any implication operand according to the rules.
#Inputs: Clause
#Outputs: Clause without any implication operand.
def solve_implication(clause):
    if isat(clause): # Atom
        return clause # There are no (<=>), done, return itself
    if isand(clause) or isor(clause): # or, and
        clause=[clause[0]]+[solve_implication(clause[1])]+[solve_implication(clause[2])] # Keep the
            symbol, recursively check the rest of the sentence
        return clause
    if isneg(clause): # Not
        clause=[clause[0]]+[solve_implication(clause[1])] # Keep the 'not', recursively check the
            rest of the sentence
        return clause
    if isimp(clause): # Get rid of the implication, recursively check the rest of the sentence
        clause=['or']+[['not']+[solve_implication(clause[1])]]+[solve_implication(clause[2])]
        return clause

# SOLVE NEGATION
#Description:Modifies a clause to stay without any negation operand not related with an atom
    according to the rules.
#Inputs: Clause
#Outputs: Clause without any negation operand not related with an atom.
def solve_negation(clause):
    if isneg(clause): # Not
        if isneg(clause[1]): # NotNot: Double negation, remove both
            clause=solve_negation(clause[1][1])
            return clause
        if isand(clause[1]): #NotAnd: Negate And (Or) and also negate both terms, recursively
            recheck the sentence
            clause=['or',['not',clause[1][1]],['not',clause[1][2]]]
            clause=solve_negation(clause)
            return clause
        if  isor(clause[1]): #NotOr: Negate Or (And) and also negate both terms, recursively
            recheck the sentence
            clause=['and',['not',clause[1][1]],['not',clause[1][2]]]
            clause=solve_negation(clause)
            return clause
    if isand(clause) or isor(clause): #And/Or: Keep the And/Or, recursively check the rest of the
        sentence
        clause[1]=solve_negation(clause[1])
        clause[2]=solve_negation(clause[2])
    return clause

# SOLVE DISJUNCTION
#Description:Applies the distributive rule in the input clause
#Inputs: Clause
#Outputs: Clause after the distibutive rule was aplied.
def solve_disjunction(clause):
    if isor(clause): #Or
        if isand(clause[1]) and isand(clause[2]): #OrAnd()And(): Apply distributive
            aux0=['and', ['or', clause[1][1], clause[2][1]], ['or', clause[1][1], clause[2][2]]]
            aux1=['and', ['or', clause[1][2], clause[2][1]], ['or', clause[1][2], clause[2][2]]]
            clause=['and', solve_disjunction(aux0), solve_disjunction(aux1)]
            if solve_disjunction(aux0) == solve_disjunction(aux1): #Simplifies, prevents clauses
                with the same atom to appear
```

```
                return solve_disjunction(aux0)
            return clause
        if isand(clause[1]): #OrAnd..: Apply distributive
            aux0=['or', clause[1][1], clause[2]]
            aux1=['or', clause[1][2], clause[2]]
            clause=['and', solve_disjunction(aux0), solve_disjunction(aux1)]
            if solve_disjunction(aux0) == solve_disjunction(aux1): #If repeated, Simplify
                return solve_disjunction(aux0)
            return clause
        if isand(clause[2]): #Or..And: Apply distributive
            aux0=['or', clause[1], clause[2][1]]
            aux1=['or', clause[1], clause[2][2]]
            if solve_disjunction(aux0) == solve_disjunction(aux1): #If repeated, Simplify
                return solve_disjunction(aux0)
            clause=['and', solve_disjunction(aux0), solve_disjunction(aux1)]
            return clause
        if isor(clause[1]) or isor(clause[2]): #OrOr, keep it, recursively check the rest of the
             sentence
            aux=solve_disjunction(clause[1])
            aux1=solve_disjunction(clause[2])
            clause=['or',aux,aux1]
            if isand(aux) or isand(aux1):
                clause=solve_disjunction(clause)
            return clause
    if isand(clause): #And, keep it, recursively check the rest of the sentence
        aux1=solve_disjunction(clause[1])
        aux2=solve_disjunction(clause[2])
        if aux1 == aux2: #If repeated, Simplify
            return aux1
        return ['and',aux1,aux2]
    return clause

#SIMPLIFY ORS, while converting to cnf clauses
#Description:Simplifies repeated conjunctions, or reduntant conjuctions
#Inputs: Clause
#Outputs:Simplified clause.
def simplify_ors(clause):
    i=0
    while i<len(clause):
        j=0
        while j<len(clause[i]):
            if isor(clause[i]): #Simplifies (A or B), (B or A) then (A or B), also (A or B), A then
                 A, also (A or A) then A
                if clause[i][1]==clause[i][2]:#A or A then A
                    clause[i]=simplify_ors(clause[i][1])
                    return clause
                auxi=i
                i=0
                while i<len(clause):
                    if clause[i] == [clause[auxi][1]] or clause[i] == [clause[auxi][2]]: #Simplifies
                         (A or B), A, then, A
                        del(clause[auxi])
                        return simplify_ors(clause) #Always recursively
                    if isor(clause[i]) and i != auxi: #Simplifies (A or B), (B or A), then, (A or B)
                        if clause[auxi][1]==clause[i][1] and clause[auxi][2]==clause[i][2]:
                            del(clause[i])
                            return simplify_ors(clause) #Always recursively
                        elif clause[auxi][1]==clause[i][2] and clause[auxi][2]==clause[i][1]:
                            del(clause[i])
                            return simplify_ors(clause) #Always recursively
                        return clause
                    i=i+1
                i=auxi
            j=j+1
        i=i+1
    return clause
```

```
# CONVERT TO CNF CLAUSES
#Description:Modifies a clause to be in a simplified form withou any operand, in a disjunction o
    conjuctions.
#Inputs: Clause
#Outputs: Clause without any operand.
def convert_clauses(clause):
    new1=[]
    convert_aux1(clause,new1) #removes ands
    new1=simplify_ors(new1) #simplify ors, before remove them
    new2=[[] for i in range(len(new1))]
    for i in range(len(new1)): #remove ors
        convert_aux2(new1[i], i,new2)
    return new2

# AUXILIAR ONE of convert to cnf clauses (remove ands from notation)
def convert_aux1(clause, new):
    if isand(clause):
        convert_aux1(clause[1], new)
        convert_aux1(clause[2], new)
    elif isneg(clause):
        new.append(clause)
    else:
        new.append(clause)
    return 0

# AUXILIAR TWO of convert to cnf clauses (remove ors from notation)
def convert_aux2(foo,i,new1):
    if isor(foo):
        convert_aux2(foo[1], i, new1)
        convert_aux2(foo[2], i, new1)
    elif isneg(foo):
        new1[i].append(foo)
    else:
        new1[i].append(foo)
    return 0

# SIMPLIFIES CNF CLAUSES TO THE IRREDUCIBLE FORM
#Description:Simplifies the CNF clause to its irreducible form.
#Inputs: Clause
#Outputs: Simplified clause.
def simplifies_one(clause):
    i=0
    while i<len(clause):
        j=0
        while j<len(clause[i]):
            if isat(clause[i]): #Simplifies repeated atoms (A or A), then, A
                auxj=j
                j=0
                while j<len(clause[i]):
                    if clause[i][auxj]==clause[i][j] and j!=auxj:
                        del (clause[i][j])
                        return simplifies_one(clause) #Always recursively
                    j=j+1
                j=auxj
            if isneg(clause[i][j]): #Simplifies Impossible, cannot be satisfied (=0)
                auxi=i
                auxj=j
                i=0
                while i<len(clause):
                    j=0
                    while j<len(clause[i]):
                        if clause[auxi][auxj][1]==clause[i][j] and (i!=auxi or j!=auxj):
                            aux=clause[auxi][:]
                            del(aux[auxj])
                            aux1=clause[i][:]
```

```
                        del(aux1[j])
                        if aux==aux1:
                            clause='Impossible, cannot satisfy both: '+str(clause[auxi])+' and
                                '+str([clause[i][j]])
                            return simplifies_one(clause) #Always recursively
                    j=j+1
                i=i+1
            i=auxi
            j=auxj
        if isneg(clause[i][j]): #Simplifies Unitary, if only clause then always satisfiable (=1)
            auxj=j
            j=0
            while j<len(clause[i]):
                if clause[i][auxj][1]==clause[i][j]:
                    aux=clause[i]
                    del(clause[i])
                    if clause==[]:
                        return 'Unitary, always satisfiable: '+str(aux)
                    return simplifies_one(clause) #Always recursively
                j=j+1
            j=auxj
        j=j+1
    i=i+1
return clause

#CNF CONVERTER FUNCTION
#Description:Modifies a clause to stay without any equivalence operand according to the rules.
#Inputs: Clause to be converted
#Outputs: step1, step2, step3, step4, step5, step6, every step of the conversion.
def CNF_converter(sentences):
    i=0
    step1=[]
    step2=[]
    step3=[]
    step4=[]
    step5=[]
    step6=[]
    while i<len(sentences):
        step1=step1+[solve_equivalence(sentences[i][:])] #Does Step1, get rid of equivalences (<=>)
        step2=step2+[solve_implication(step1[i][:])] #Does Step2, get rid of implications (=>)
        step3=step3+[solve_negation(step2[i][:])] #Does Step3, move negations inwards (not)
        step4=step4+[solve_disjunction(step3[i][:])] #Does Step4, apply the distributive to
            disjunctions (or)
        step5=step5+[convert_clauses(step4[i][:])] #Does Step5, put CNF as a set of disjunctions
        step6=step6+[simplifies_one(step5[i][:])] #Does Step6, simplifies [[not, A],A] to 1
        i=i+1
    return step1, step2, step3, step4, step5, step6 #Returns the sentences in all the steps of
        conversion

# PRINT SENTENCES FUNCTION, runs all the lines and prints it
#Input: mylist,
#Output: NONE
def print_sentences(mylist, f):
    i=0
    while i<len(mylist[:]):
        f.write('Nr'+str(i+1)+': '+str(mylist[i])+('\n'))
        print('Nr'+str(i+1)+': '+str(mylist[i]))
        i=i+1
    return

# PRINT STEPS FUNCTION, explains what list is printing and prints it
#Input: explanation of the step, the step itslef
#Output: NONE
def print_steps(Explanation, mylist, f):
    print(Explanation)
    f.write(Explanation+'\n')
```

```
    print_sentences(mylist, f) #Invoces print_sentences to print list
    return
```