Instituto Superior Técnico

# IASD

Inteligência Artificial e Sistemas de Decisão

## Laboratório Nr. 3

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

12 de Dezembro de 2014

# Contents

# 1 Introdução

Este relatório apresenta a resolução do problema proposto como o terceiro laboratório de IASD, no qual se pretende criar um "Theorem Prover". Desta forma, o objectivo deste laboratório é um complemento ao anterior laboratório, cujo objectivo era criar um programa de conversão para o formato CNF (Clausal normal form), onde agora o objectivo é desenvolver um algoritmo que prove a veracidade de uma dada sentence no formato CNF, usando resolução, factorização e a preferência unitária.

O programa tem de ser capaz de receber comandos do utilizador, e consoante o desejado, mostrar e listar a frase lógica e o KB (Knowledge Based), mostrando se a mesma é verdadeira ou falsa, e ainda todos os passos intermédios necessários para a realizar a prova. Tal como no anterior laboratório, as frases lógicas são recebidas por via de um ficheiro de dados, sendo posteriormente toda a informação pedida pelo utilizador gravada noutro ficheiro de dados.

# 2 Algoritmo

À semelhança do anterior laboratório, 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 três ficheiros de código, separando a função mãe `lab3_main.py`. que contém o menu e invoca as funções necessárias, da função com as funções e algoritmos de conversão usados no laboratório passado e que serão necessarios para a resolução deste `fu.py`, assim como as funções e algoritmos para resolução do teorema em si `fu2.py`. É de salientar que este algoritmo apresenta solidez (soundness), propriedade indispensável em algoritmos dedutivos.

A primeira acção executada pelo algoritmo é a conversão para a forma CNF, através do algoritmo desenvolvido no laboratório anterior. Executando-se a leitura do ficheiro que contém o Knowledge Based, sendo toda a informação armazenada e convertida para o formato CNF, de uma forma correcta e eficiente.

Adicionalmente, é efectuada a leitura do ficheiro que contém alpha, sendo essa informação negada, e convertida para o formato CNF e de seguida adicionada ao topo de uma nova lista juntamente com o Knowledge Based. Através desta negação pretende-se executar uma prova por contradição(proof by contradiction) onde se no fim das etapas de resolução for gerada uma contradição(empty clause) então prova-se o contrário($\alpha$ no presente caso). Desta forma, a base para a prova do teorema $KB \vdash alpha$ está desenvolvida, sendo de salientar que a maneira como esta acção é feita é um elemento chave para a execução do restante algoritmo.

Após o KB e o alpha serem armazenados em forma de lista, e consoante o pedido do utilizador, é possível listar ambos ou proceder à prova do teorema via resolução, usando a lista criada acima referida. Este processo consiste numa sequência de três 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 prova. Este algoritmo apresenta-se robusto, aceitando diversas cláusulas para o KB ou mesmo alpha, sendo que neste caso, o algoritmo efectua a prova de alpha cláusula a cláusula, onde para alpha ser provado, todas as linhas de alpha têm de ser provadas com sucesso. Agora são apresentados os principais passos presentes no algoritmo:

1. Simplificar, removendo as expressões lógicas repetidas ou contidas noutras expressões com mais restrições(factorização(factoring)), como por exemplo tendo $A \wedge A$ fica $A$ ou no caso de uma expressão com mais restrições $A \wedge A \vee B$ fica simplesmente $A$. Este primeiro passo é implementado na função `remove` onde uma lista de clauses entra na função é duplicada e com um sequência de ciclos e se verifica a existência de expressões lógicas repetidas e eliminadas. Nesta função faz-se também uso da recursividade de modo a conseguir garantir a saída correcta desta função;

2. Ordenar expressões lógicas, da mais pequena para a maior, ou seja, começando na com menos literais e terminando na com mais literais. Desta forma, estamos a garantir a aplicação da regra 'preferência unitária', por exemplo $A \vee B \wedge C \vee D \vee A \wedge B$ vai ficar $B \wedge A \vee B \wedge C \vee D \vee A$. Esta acção é executada pela função `order` que através de dois ciclos, re-ordena as clausulas na lista consoante o seu tamanho como exemplificado anteriormente;

3. Procurar pares, começando no início (preferência unitária), procura pares de literais complementares $A$ e $\neg A$ , removendo ambas as expressões e criando uma nova expressão com o remanescente de ambas as expressões lógicas (método da resolução), como por exemplo $A \vee B \wedge \neg A \vee B$ fica $B$. Este passo está implementado na função `resolution` que através de uma sequência ciclica verifica todos os literais presentes, encontrando aqueles que são complementares criando uma nova cláusula com o remanescente de ambos. Este invoca ainda as funções acima referidas por forma a garantir que a lista se mantém ordenada (preferência unitária) e na sua forma simplificada.

Este último passo, que invoca os outros dois passos explicados, é repetido consecutivamente para provar o teorema $KB \vdash alpha$. Este processo cíclico é interrompido, por um lado, quando obtivermos uma expressão lógica vazia (teorema é provado com sucesso, a negação de alpha levou a uma contradição, portanto alpha é provado), ou, por outro lado, quando não há mais pares de literais complementares (não é possível provar o teorema dado este alpha).

# 3 Resultados

Através deste código em python, foi assim obtido um algoritmo que resolve o teorema $KB \vdash alpha$, para um dado KB e alpha. Graças ao trabalho anteriormente desenvolvido (segundo laboratório), é possível a introdução das expressões em lógica proposicional , sendo estas convertidas para o formato CNF, por forma a prosseguir a resolução do teorema. Em anexo, são apresentados vários exemplos de teoremas resolvidos onde se escolhe um determindado $\alpha$ para ser provado num $KB$. Pode-se assim verificar o correcto funcionamento do algoritmo, através de uma abordagem 'step-by-step', para uma análise e verificação mais clara.

# 4 Conclusão

Com a implementação deste algoritmo, e após todos os testes efectuados para verificação do mesmo, foi possível concluir que o laboratório foi desenvolvido com sucesso. O facto de o laboratório anterior ter abordado um conversor para a forma CNF foi essencial para o desenrolar do presente laboratório. Todas as funções desejáveis e todos os requisitos foram implementados, onde todos os passos do algoritmo, que correspondem às diferentes partes do método (regras de factorização, preferência unitária e resolução) funcionam correctamente.

# 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

In this section several examples tried to validate the algorithm developed are presented, including a step by step resolution in the end:

- Situation one:
  - Alpha:
    ('not', 'C')
  - KB:
    ('or', ('or', ('not', 'E'), 'D'), 'T')
    ('or', ('not','T'),'C')
    ('or',('or', ('not', 'D'), 'E'), 'I')
    ('or',('or', ('not', 'D'), 'I'), 'G')
    ('or', ('or', ('or', ('not', 'D'), 'C'), 'G'), 'T')
    ('or', ('not', 'I'), ('not', 'G'))
    ('or', ('not','E'),('not', 'D'))
  - Result:Nothing Can Be Concluded.
- Situation two:
  - Alpha:
    ('not', 'A')
  - KB:
    ('A')
  - Result: Nothing Can Be Concluded.
- Situation three:
  - Alpha:
    ('C')
  - KB:
    ('or','C',('or',('not','E'),'D'))
    ('or',('not','T'),'C')
    ('or',('and',('or','E',('not','G')),('not','D')),'I')
    ('or','T',('or','C',('or',('not','D'),'G')))
    ('not',('not','C'))
    ('<=>',('not','I'),('not','G'))
    ('or',('not','D'),('not','E'))
  - Result:Alpha Proved.
- Situation four:
  - Alpha:
    ('<=>', 'Poluidor',('not', 'Ciclista'))
  - KB:
    ('<=>', ('not', 'Ciclista'), 'Poluidor')
  - Result:Alpha Proved.
- Situation five:
  - Alpha:
    ('or', ('not','A'),'A')

  - KB: None
  - Result: Alpha Proved.

- Situation six:
  - Alpha:
    ('or', ('not','A'),'A')
  - KB:None
  - Result:Alpha Impossible,

- Situation seven:
  - Alpha:
    ('or', ('not','A'),'A')
    ('B')
  - KB:
    ('or', ('not','A'),'A')
    ('and', ('or', 'B', 'C'), ('not', 'A'))
    ('or', ('or', 'A', 'B'), 'C')
    ('and', 'D', 'C')
  - Result:Nothing can be concluded.

- Situation eight:
  - Alpha:
    ('or', ('not','A'),'A')
    ('B')
  - KB:
    ('or', ('not','A'),'A')
    ('and', 'A', 'C')
    ('or', ('or', 'A', 'B'), 'C')
    ('and', 'B', 'A')
  - Result:Alpha Proved.

- Situation nine:
  - Alpha:
    ('A')
    ('B')
  - KB:
    ('and', ('not','A'),'A')
  - Result:KB given invalid, Impossible clause included.

The situation 4 will now be presented in a step by step approach.

- Alpha:
  ('<=>', 'Poluidor',('not', 'Ciclista'))

- KB:
  ('<=>', ('not', 'Ciclista'), 'Poluidor')

- Steps: Before Conversion, Step:0 (line in alpha)
  Nr1: ['Poluidor', ['not', 'Ciclista']]
  Nr2: ['Ciclista', ['not', 'Poluidor']]
  Nr3: ['Ciclista', 'Poluidor']
  Nr4: [['not', 'Poluidor'], ['not', 'Ciclista']]
  Simplified, Step:0 (line in alpha)
  Nr1: ['Poluidor', ['not', 'Ciclista']]
  Nr2: ['Ciclista', ['not', 'Poluidor']]
  Nr3: ['Ciclista', 'Poluidor']
  Nr4: [['not', 'Poluidor'], ['not', 'Ciclista']]
  Ordered, Step:0 (line in alpha)

Nr1: ['Poluidor', ['not', 'Ciclista']]
Nr2: ['Ciclista', ['not', 'Poluidor']]
Nr3: ['Ciclista', 'Poluidor']
Nr4: [['not', 'Poluidor'], ['not', 'Ciclista']]
Resolution for all possible clauses, Step:0 (line in alpha)
(Finds all the complementary pairs)
Nr1: ['Ciclista', ['not', 'Poluidor']]
Nr2: [['not', 'Poluidor'], ['not', 'Ciclista']]
Nr3: ['Poluidor']
Resolution for all possible clauses, Step:0 (line in alpha)
(Finds all the complementary pairs)
Nr1: []
The theorem prover has ended successfully
Empty Clause -> Alpha Proved!

# 6 Código Python

Listing 1: File lab3_main.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 #From last assignment, in order to put in the CNF form
import fu2 #To invoce the functions to prove the theorem
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_kb= fu.open_file('kb10.txt') #Reads info given in input file, by line
read_alpha= fu.open_file('alpha10.txt') #Reads info given in input file, by line
kb=fu.file_to_list(read_kb) #Convert lines of the file into a list
alpha=fu.file_to_list(read_alpha) #Convert lines of the file into a list
f = open('output_file.txt', 'w') #Initiate Damp File, that will store everything
step1, step2, step3, step4, step5, kb_cnf=fu.CNF_converter(kb) #Conversion Step by Step
step1, step2, step3, step4, step5, alpha_cnf=fu.CNF_converter(alpha) #Conversion Step by Step

flag=1
while flag==1: #MENU : CHOOSE AN OPTION
    print('Artificial Intelligence and Decision Systems (IASD)')
    print('Assignment #3 - Version: Python 3.4.1')
    print('Eduardo Silva nr 69916 - MEAer')
    print('Jose Pereira nr 70369 - MEAer')
    print('\nOptions:\n\t1: List KB and alpha before theorem prover')
    print('\t2: Theorem Prover - List it Step-by-Step\n\t3: Exit')
    opt1=input('Choose an option [1, 2 or 3]\n')
    initial_time = time() #Set initial time

    if opt1=='1': #If chooses '1', prints kb and alpha
        fu.print_steps('List kb:', kb_cnf, f) #Print Sentences
        fu.print_steps('List alpha:', alpha_cnf, f) #Print Sentences

    elif opt1=='2': #If chooses '2', proves theorem step-by-step
        need_to_prove, result=fu2.check_inputs(alpha_cnf, kb_cnf,f)
        if need_to_prove==1:
            step=0
            while step<len(alpha) and result!=1:
                not_alpha='not', alpha[step]
```

```
            step1, step2, step3, step4, step5, not_alpha_cnf=fu.CNF_converter([not_alpha])
                #Conversion Step by Step
            done=0
            if not_alpha_cnf[0][0]=='I':
                del(not_alpha_cnf)
                done=1
            elif not_alpha_cnf[0][0]=='U':
                del(not_alpha_cnf)
                result=1
                done=1
            if done!=1:
                kb_cnf=not_alpha_cnf+kb_cnf
                foo=0
                lista=[]
                while foo<len(kb_cnf):# Puts all the KB in the same line/clause
                    lista=lista+kb_cnf[foo]
                    foo=foo+1

                print('Before Conversion, Step:'+str(step)+' (line in alpha)')
                f.write('\nBefore Conversion Step:'+str(step)+' (line in alpha)\n')
                fu.print_sentences(lista, f) #Prints whole list (KB and not alpha) before
                    conversion
                lista=fu2.remove(lista) #Simplifies list
                print('Simplified, Step:'+str(step)+' (line in alpha)')
                f.write('\nSimplified, Step:'+str(step)+' (line in alpha)\n')
                fu.print_sentences(lista, f) #Prints simplified list (KB and not alpha)
                lista=fu2.order(lista) #Puts list in order
                print('Ordered, Step:'+str(step)+' (line in alpha)')
                f.write('\nOrdered, Step:'+str(step)+' (line in alpha)\n')
                fu.print_sentences(lista, f) #Prints organized list (KB and not alpha)
                check_changes=[]
                result=1
                while lista!=check_changes and result!=0: #Performs resolution
                    check_changes=lista[:]
                    lista,result=fu2.resolution(lista)
                    print('Resolution for all possible clauses, Step:'+str(step)+' (line in
                        alpha)\n(Finds all the complementary pairs)')
                    f.write('\nResolution for all possible clauses, Step:'+str(step)+' (line in
                        alpha)\n(Finds all the complementary pairs)\n')
                    fu.print_sentences(lista, f) #Prints list (KB and not alpha) after
                        performing resolution
            step=step+1
    print('The theorem prover has ended successfully')
    f.write('\nThe theorem prover has ended successfully')
    if result==0:
        print('Empty Clause -> Alpha Proved!')
        f.write('\nEmpty Clause -> Alpha Proved!\n')
    if result==1:
        print('Cannot Improve More -> Nothing Can Be Conclude!')
        f.write('\nCannot Improve More -> Nothing Can Be Conclude!\n')
    if result==2:
        print('KB given invalid, Impossible clause included!')
        f.write('\nKB given invalid, Impossible clause included!\n')
    flag=0

elif opt1=='3': #If chooses '3' Exits
    flag=0

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('-----------------------------------------------------------')
f.write('\nComputational Time: '+str(computational_time)+' seconds')
```

```
    f.write('\n-------------------------------------------------------\n')
    if flag==0:
        f.close()
```

Listing 2: File fu2.py

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

import copy #To copy lists without using pointers
import fu #From last assignment, in order to use the function "simplifies_one"
from time import time #To count computational time

#Check Inputs
#Description: Check if there is any mistake in the given inputs
#Inputs: KB and alpha
#Outputs: Flag signing the need to prove the theorem
def check_inputs(alpha_cnf, kb_cnf,f):
    result=-1
    need_to_prove=1
    foo=0
    while foo<len(alpha_cnf):
        if alpha_cnf[foo][0]=='I':
            result=1
            need_to_prove=0
            print('Alpha Impossible')
            f.write('\nAlpha Impossible\n')
        if alpha_cnf[foo][0]=='U':
            if result!=1:
                result=0
            del(alpha_cnf[foo])
            foo=foo-1
            if alpha_cnf==[]:
                need_to_prove=0
        foo=foo+1
    foo=0
    while foo<len(kb_cnf):
        if kb_cnf[foo][0]=='I':
            need_to_prove=0
            result=2
        if kb_cnf[foo][0]=='U':
            del(kb_cnf[foo])
            foo=foo-1
        foo=foo+1
    return need_to_prove, result
#Remove
#Description: Remove less constrained and repeated clauses
#Inputs: Whole list (KB and not alpha)
#Outputs: List simplified
def remove(clause):
    for i in range(len(clause)): #all clauses, clause[i] is more restrict clause
        other=[j for j in range(len(clause))]
        other.pop(i)
        for j in other: #all other clauses, clause[j] more general clause
            foo=0
            for k in range(len(clause[i])):
                foo+=clause[j].count(clause[i][k])
            if foo==len(clause[i]): #clause[j] is unnecessary
                clause.pop(j)
                clause=remove(clause) #recursive to ensure that all contrained /repeated clauses
                    are removed.
                return clause
    return clause

#Order
```

```
#Description:Ordenates the list with clauses to put the smaller elements as the first elements of
    the list
#Inputs: Whole disorganized list (KB and not alpha)
#Outputs: Organized list, from smaller to larger number of literals (literals in the first
    elements)
def order(clause):
    ordered=[]
    foo=len(clause)
    while len(ordered)<foo:
        value=len(clause[0])
        entry=0
        for i in range(len(clause)):
            if value>len(clause[i]):
                value=len(clause[i])
                entry=i
        ordered.append(clause[entry])
        clause.pop(entry)
    return ordered

#Find Pairs
#Description: Finds Complementary Pairs (!A and A) in different clauses
#Inputs: Whole list (KB and not alpha) and a position in the list (i and j)
#Outputs: clauses with the complementary pairs
def find_pairs(lista,i,j):
    for i2 in range(len(lista)):
        for j2 in range(len(lista[i2])):
            if lista[i][j][0]=='not':
                if [lista[i2][j2]]==lista[i][j][1:]:
                    return i2,j2,1
            if lista[i2][j2][0]=='not':
                if [lista[i][j]]==lista[i2][j2][1:]:
                    return i2,j2,1
    return 0,0,0

# Resolution
#Description: Performs Resolution to all the pairs found (Also organize and simplify clauses)
#Inputs: Whole list (KB and not alpha)
#Outputs: Whole list (KB and not alpha) after resolution
def resolution(lista): #choose following unit preference
    lista=order(lista) #Order clauses
    i=0
    while i<len(lista):
        j=0
        while j<len(lista[i]):
            i2,j2,flag=find_pairs(lista,i,j)
            if flag==1: #Complementary Literals found
                aux1=list(lista[i])
                aux2=list(lista[i2])
                aux1.pop(j)
                aux2.pop(j2)
                if aux1==[] and aux2==[]:
                    new=[]
                elif aux1==[]:
                    new=aux2
                elif aux2==[]:
                    new=aux1
                else:
                    new=aux1
                    for k in range(len(aux2)):
                        new.append(aux2[k])
                lista.append(new)
                foo=0
                while foo<len(lista):
                    lista[foo]=fu.simplifies_one([lista[foo]])
                    if lista[foo][0]=='I':
                        return [],0
```

```
                if lista[foo][0]!='U':
                    lista[foo]=lista[foo][0]
                else: #If finds unitary condition remove it from the list
                    del(lista[foo])
                    if lista==[]: #If its the only condition theorem shouldnt be removed
                        lista=['Unitary']# And theorem prover must end
                        return lista,1
                foo=foo+1
            lista=remove(lista) #Remove less constrained and repeated clauses
            if new==[] or lista==[]: #If so, empty clause -> alpha proved
                return lista,0
        j=j+1
    i+=1
return lista,1
```

Listing 3: File fu.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 os

#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):
    if os.stat(file).st_size != 0:
        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]
    else:
        read_file=[]
```

```
    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
```