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 ('string', '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, uma função auxiliar foi criada (checkoffset), que identifica o tipo de regra em causa, retornando o número de argumentos da mesma (e.g. 3 argumentos, o símbolo equivalente mais as duas 'sentences').

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, procuramos 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 {!A, E},{B, E}. Para tal, as negações são introduzidas dentro do próprio argumento, os argumentos das disjunções são acoplados, depois de voltar a invocar recursivamente a função e por último, a conjunção mantém-se sendo a função novamente invocada para os seus argumentos;

——————— REVI ATÉ AQUI———————

Após estes cinco passos foi conseguida a conversão do conjunto de frases lógicas num conjunto de cláusulas. Sendo ainda necessário eliminar possíveis repetições (feito na disjunction) e executar simplificações. Com esse objectivo realizaram-se os seguintes passos:

1. Eliminar impossiveis, condições unitarias etc etc

Referir menu criado e diversas opções

Escrita dos resultados

# 3 Conclusão

Successo??
foram todas as funçoes implementadas??
dificuldades encontradas??
pontos a melhorar??

# 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*