Inteligência Artificial e Sistemas de Decisão
1.º semestre - 2017/2018
Propositional logic reasoner

Rui Figueiredo (78247) , Alexandre Candeias (78599)

I. Implementação

A. Converter

A implementação realizada para o conversor está dividida em três passos principais. Numa primeira fase o ficheiro é lido e é construida uma árvore binária em que os nós podem tomar os valores dos literais, negados ou não negados e os operandos $\vee, \wedge$. Numa segunda fase a árvore é convertida para a forma de CNF, *conjuntion normal form*, para obter esta forma basta aplicar a propriedade distributiva na árvore inicial. A terceira fase consiste em percorrer a árvore construida e retirar todas as *sentences*.

1) Construção da árvore: A construção da árvore é feita de forma recursiva através da função R(input), o retorno da função para cada um dos valores de input vai de acordo com o que está apresentado na tabela I. O valor representado na coluna 2 representa (valor do nó, filho direito, filho esquerdo). Os valores de A e B podem ser qualquer expressão válida na linguagem definida para representação das proposições. Por exemplo a proposição $\neg(A \wedge B) => \neg(\neg C \vee \neg D) \equiv (A \vee B) \wedge (C \vee D)$ dá origem à árvore apresentada em 1.

input	R(input)
(and,A,B)	(∧,R(A),R(B))
(or,A,B)	(∨,R(A),R(B))
(=,A,B)	(∨,R((not,A)),R((not,B)))
(=,A,B)	(∧,R((=,A,B)),R(=,B,A))
(not,(and,A,B))	(∨,R(('not',A)),R((no',B)))
(not,(or,A,B))	(∧,R((not,A)),R((not,B)))
(not,(=,A,B))	(∨,R(A),R(('not',B)))
(not,(=,A,B))	(∨,R((not,(=>,A,B)),R((not,(=>,B,A))))
(not,(not,A))	R(A)
literal	(literal,-,-)
(not,literal)	((not,literal),-,-)

TABLE I
RETORNOS DA FUNÇÃO RECURSIVA DE CONTRUÇÃO DA ÁRVORE

2) Aplicação da Propriedade distributiva: Após a construção da árvore, a mesma é percorrida através de uma BFS, se durante a busca encontramos um nó com o valor $\vee$ que tem um filho com o valor $\wedge$ isso significa que temos de aplicar a propriedade distributiva entre estes dois nós.

A propriedade distributiva é aplicada, segundo o pseudo-código apresentado em 1, é de notar que a função Node(value,leftchild,rightchild) cria um novo nó da árvore binária com o valor value e com o filhos leftchild e rightchild. De uma forma visual também podemos observar as figuras 1 e 2, a figura 2 é o resultado de aplicar a propriedade distributiva na árvore da figura 1 entre os nós 1 e 2, sendo que o nó 1 é o nó da raiz e numeramos os filhos da esquerda para a direita.

3) Construção das sentences: Após a aplicação da propriedade distributiva a árvore obtida já se encontra na forma de CNF e desta forma podemos já percorrer a árvore de modo a extrair todas as sentences.

Ao percorrer a árvore, ao encontramos um nó $\wedge$ que têm um filho em que o valor não é $\wedge$ isso significa que esse filho é a raiz de uma nova sentence, esta raiz é então guardada numa lista para posteriormente ser impressa com o formato desejado. Durante a conversão também é realizada uma primeira simplificação no conjunto das *sentences* em que se remove *sentences* repetidas e tautologias.

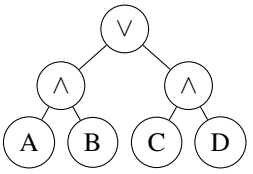

Fig. 1. Árvore Inicial

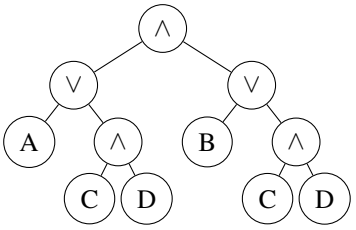

Fig. 2. Árvore após aplicar a propriedade distributiva entre os nós 1 e 2 da árvore da figura 1

Algorithm 1: Algoritmo para aplicação da propriedade distributiva a uma árvore com a apresentada na figura 1

```
   input : andnode,ornode
1  begin
2      if ornode.left == andnode then
3          otherson := ornode.right
4      else
5          otherson := ornode.left
6      ornode.value = ∧
7      ornode.left = Node(∨,otherson,andnode.left)
8      ornode.right = Node(∨,otherson,andnode.right)
```

B. Prover

O algoritmo utilizado para a *resolution* utilizada no prover é o disponibilizado no livro adotado pela disciplina e apre-

sentado em 2. Em cada ciclo do algoritmo do prover é ainda executada um simplificação na *knowledge base* seguindo as regras abaixo:

1) Uma clausula pode ser removida se conter um literal que não é complementar com nenhum outro literal nas restantes clausulas.
2) Uma clausula que contêm um literal e a sua negação é uma tautologia e então pode ser removida.
3) Uma clausula que é implicada por outra pode ser removida, isto é se uma clausula é um subset de outra então a maior pode ser removida
4) Se um literal ocorrer mais que uma vez numa clausula então a cópia desse literal pode ser eliminada.

Estas regras correspondem à função simplify no pseudocódigo apresentado. Na implementação garantiu-se que a ordem de resolução seguia a *unit preference strategy*, ou seja, as *clauses* são analisadas por ordem cresceste de número de literais.

Durante o desenvolvimento do programa encontramos mais uma simplificação que podia ser feita. Inicialmente na aplicação da resolução efetuávamos todas as combinações possíveis entre duas frases, no entanto chegamos à conclusão que apenas teríamos de efetuar em ordem à primeira variável que encontrássemos.

Pensando no exemplo das frases $s_1 = \{A,B,C,D\}$ e $s_2 = \{\neg A,\neg B,C,D\}$, para este exemplo existem duas combinações, $A,\neg A$ e $B,\neg B$, para a primeira obteríamos $\{\neg B,B,C,D\}$ e da segunda $\{\neg A,A,C,D\}$. Facilmente se pode observar por este exemplo que sempre existe mais que uma combinação entre duas frases, a frase obtida por resolução é uma tautologia e logo que vai ser eliminada na simplificação.

Algorithm 2: Algoritmo base do *prover*

```
input : KB
output: true or false
begin
    clause := KB
    new := {}
    while True do
        foreach pair of clauses C_i,C_j in clauses do
            resolvents := Resolve(C_i,C_J)
            if resolvents contains the empty clause then
                return True
            new := new ⋃ resolvents
        if new ⊆ clauses then
            return False
        clauses := clauses ⋃ new
        simplify(clauses)
```

II. Resultados

A. test_converter.txt

De modo a testar o funcionamento do conversor criou-se um ficheiro de teste que continha todas as operações básicas que poderiam ocorrer. O ficheiro encontra-se nos ficheiros anexados ao código com o nome *test_converter.txt*, o resultado da conversão vai de acordo com a previsão teórica.

B. Teste do Prover

A partir dos exemplos encontrados em `http://www.danielclemente.com/logica/dn.en.html` foi construido um conjunto de ficheiros de teste num total de 22. Os ficheiros são enviados em anexo ao código e o nome é do tipo fpXX.txt.

Para correr este e todos os outros testes basta executar **python3 test.py**. O programa test.py foi criado com a ferramenta de testes unitários do python com o intuito de uma forma fácil pode correr os testes sempre que havia alterações ao código.

C. Ficheiros fornecidos pelo corpo docente

O teste de funcionamento do *prover* foi efetuado com base nos ficheiros fornecidos pelo corpo docente, os resultados são apresentados na tabela II.

Durante a fase de testes observou-se que após a inclusão das simplificações enumeradas anteriormente os ficheiros {p1,p2,p3,sentences}.txt viam logo à partida reduzidas as suas clauses para o conjunto vazio, $KB = \{\}$ o que permitia logo chegar a uma conclusão sem sequer ter de aplicar nenhuma vez a resolução entre clausulas. Já para o ficheiro p4.txt após a primeira simplificação o knowledge base ficava $KB = \{\{\neg C\},\{C\}\}$, vemos também que com uma iteração da resolução o teorema é provado.

Os valores para o número de clauses à entrada do *prover* e o número de clauses após primeira simplificação.

Teste File	Input Clauses	Clauses after first simp	Output
sentenses.txt	4	0	False
trivial.txt	4	3	True
p1.txt	10	0	False
p2.txt	4	0	False
p3.txt	11	0	False
p4.txt	13	2	True

TABLE II
RESULTADOS PARA OS FICHEIROS FORNECIDOS PELO CORPO DOCENTE.

III. Conclusões

Em problemas deste tipo o conjunto de simplificações que podemos assumir é o fator mais preponderante no desempenho do algoritmo. Outra coisa que simplifica muito a resolução de problemas é a forma como o problema é representado na implementação feita, por exemplo a representação em árvore das frases, e a recursividade na sua análise permitiu uma implementação muito simples do algoritmo de conversão.

Por fim seria interessante comparar a performance deste método em relação a uma abordagem usando o algoritmo de DPLL para provar a *unsatisfiabilty* da frase CNF. De modo a testar as duas abordagens teriam de ser utilizados *datasets* com um maior número de frases e variáveis dos que os que foram utilizados neste trabalho.