UNIVERSIDADE DE SÃO PAULO
CAMPUS DE SÃO CARLOS
INSTITUTO DE CIÊNCIAS MATEMÁTICAS DE SÃO CARLOS
I.C.M.S.C.

# Método sintático de prova de teoremas
## Algorítmo de Wang

MONARD, M.C. ; NICOLETTI, M.C.

Nº 62

Notas do ICMSC - USP

ISSN 0103-2577

# Método sintático de prova de teoremas
# Algorítmo de Wang

MONARD, M.C. ; NICOLETTI, M.C.

Nº 62

São Carlos (SP)
1990

# Método Sintático de Prova de Teoremas

# Algoritmo de Wang

Maria Carolina Monard [1]
Universidade de São Paulo / ILTC
Instituto de Ciências Matemáticas de São Carlos
Departamento de Ciências de Computação e Estatística

Maria do Carmo Nicoletti
Universidade Federal de São Carlos / ILTC
Departamento de Computação

## Sumário

A Lógica é de significativa importância na área de Inteligência Artificial para resolver problemas nos quais o conhecimento pode ser axiomatizado. Há, portanto, um grande interesse em implementações eficientes de Provadores Automáticos de Teoremas.
Neste trabalho é apresentado o método sintático de prova de teoremas baseado no algoritmo de Wang; são analisadas duas conhecidas implementações deste método — feitas na linguagem de programação lógica Prolog — e, com base em estudos detalhados e análises de desempenho, é proposta uma terceira implementação, também em Prolog, sensivelmente mais eficiente que as outras duas. Medições de tempo de execução são mostradas de maneira a evidenciar a maior eficiência da implementação proposta.

[1]Trabalho realizado com auxílio do CNPq e ILTC - Instituto de Lógica Filosofia e Teoria da Ciência

# Conteúdo


1 Introdução 1

2 Considerações Preliminares sobre Prova de Teoremas 2

3 Exemplo de Prova de Teorema via Tabela-verdade 3

4 Método Sintático de Prova de Teorema 5

5 Algoritmo de Wang 6

5.1 Regras de Transformação . . . . . 7

5.2 Regras de Parada . . . . . 9

6 Implementações do Algoritmo de Wang 10

6.1 Implementação de L. Pereira . . . . . 12

6.1.1 Listagem do Programa . . . . . 13

6.1.2 Exemplos de Execução . . . . . 15

6.2 Implementação de Bratko . . . . . 24

6.2.1 Listagem do Programa . . . . . 27

6.2.2 Exemplos de Execução . . . . . 30

6.3 Implementação de MCM-MCN . . . . . 36

6.3.1 Listagem do Programa . . . . . 37

6.3.2 Exemplos de Execução . . . . . 39

7 Medições de Tempos de Execução das Implementações e Análise dos Resultados 46

7.1 Teoremas e não Teoremas usados nas Medições . . . . . 46

7.2 Tempos de Execução dos Teoremas . . . . . 48
7.3 Tempos de Execução dos Não Teoremas . . . . . 51

**8 Conclusões** **53**

# 1 Introdução

Lógica e Prova Automática de Teoremas são de significativa importância na área de Inteligência Artificial — a primeira por ser uma linguagem na qual se pode expressar problemas e, a segunda, por ser uma possível forma de obter suas soluções.

São vários os pesquisadores de Inteligência Artificial que acreditam que a Lógica Matemática fornece a melhor linguagem para a representação de conhecimento. Muito embora isso seja discutível, ninguém pode negar que os métodos da Lógica são poderosos e devem ser cuidadosamente analisados por pessoas interessadas na solução automática de problemas.

O Cálculo Proposicional - CP - embora aplicável a um número restrito de problemas, fornece uma ferramenta simples com a qual os conceitos básicos de prova automática de teoremas podem ser ilustrados.

Neste trabalho é apresentado um método sintático para a prova automática de teoremas do CP, proposto originalmente por H.Wang [Wang 60],[Wang 64].

Duas conhecidas implementações desenvolvidas na linguagem de programação lógica Prolog são discutidas, abordando as estruturas de dados utilizadas bem como a eficiência de execução, procurando evidenciar os aspectos positivos e negativos de cada uma delas.

Com base nesta discussão, é proposta uma terceira implementação em Prolog do método de Wang, onde diversas otimizações foram realizadas.

A fim de comparar a eficiência das implementações, foi selecionado um conjunto de 50 teoremas e 21 não teoremas, e medidos os tempos de execução das tres implementações. As medidas obtidas evidenciam a superioridade da implementação proposta, que, em média, executa aproximadamente duas vezes mais rápido do que a mais eficiente das outras duas implementações.

O trabalho está organizado da seguinte forma:

Na seção 2 são feitas algumas considerações sobre o método semântico de prova de teoremas e na seção 3 mostramos um exemplo de prova de teorema via tabela-verdade, evidenciando o problema de explosão combinatória que eventualmente este método pode provocar, quando automatizado.

Na seção 4 é descrito o método sintático de prova de teoremas que, devido às suas características, é mais apropriado para ser tratado pela máquina.

Na seção 5 é apresentado o método sintático de prova de teoremas proposto por H. Wang e na seção 6 são apresentadas tres implementações deste algoritmo; a primeira proposta por L. Pereira, a segunda por I. Bratko e a terceira, por nós.

Na seção 7 são mostrados os tempos de execução das tres implementações e análise dos resultados obtidos. Finalizando o trabalho, são apresentadas conclusões e uma proposta para pesquisa futura na linha de prova automática de teoremas.

# 2 Considerações Preliminares sobre Prova de Teoremas

Sejam $p_1,p_2,\ldots,p_n$ e c proposições para as quais:

$$p_1 \wedge p_2 \wedge ,\ldots, \wedge p_n \rightarrow c$$

é uma tautologia. Pode-se então dizer que a conjunção dos $p_i$ logicamente implica c. As proposições $p_1,p_2,\ldots,p_n$ são chamadas de *premissas* e a proposição c de *conclusão* do *silogismo* **se** $p_1,p_2,\ldots,p_n$ **então c**. Neste caso, **se** $p_1, p_2,\ldots,p_n$ **então c** é um *teorema* e escreve-se:

$$p_1,p_2,\ldots,p_n \Rightarrow c$$

No caso da conjunção dos $p_i$ não implicar logicamente em c, não se tem um teorema e este fato é notado por:

$$p_1,p_2,\ldots,p_n \not\Rightarrow c$$

Uma forma de determinar se um silogismo é ou não um teorema é através de sua tabela-verdade. O silogismo será um teorema se o valor-verdade da implicação for v. A verificação do valor-verdade de um silogismo através de sua tabela-verdade é conhecido como *método semântico de prova de teorema.*

Devido à maneira como a implicação $\rightarrow$ é definida, é necesssário apenas verificar as instâncias nas quais todas as premissas $p_1,p_2,\ldots,p_n$ são verdade. Se a conclusão c for verdade nessas instâncias, a implicação será uma tautologia. A técnica de verificar apenas as instâncias nas quais as premissas têm valor verdade é chamada de *método forward chaining.*

Pode-se também verificar o valor-verdade de um silogismo, se for possível garantir que para cada instância em que a conclusão for falsa, pelo menos uma das premissas é também falsa. Se isto acontecer, a implicação é uma tautologia. Este método de verificação de um silogismo é chamado de *método backward chaining*.

Para a verificação do valor-verdade de um silogismo utilizando-se o método *forward chaining*, parte-se das premissas e, através do exame de seus valores-verdade, tenta-se chegar à conclusão. Quando se utiliza o método *backward chaining* com o mesmo propósito, inicia-se com a conclusão e, a partir desta, investiga-se os valores-verdade das premissas.

# 3 Exemplo de Prova de Teorema via Tabela-verdade

Sejam os seguintes enunciados assumidos como verdadeiros — premissas:

1. se *Maria come doce,* então *Maria engorda*
2. se *Maria engorda,* então *Maria faz regime*;
3. *Maria* não *faz regime.*

e que a partir deles se queira saber se: *Maria come doce.*

Se o Cálculo Proposicional - CP - for utilizado para a verificação da validade do enunciado *Maria come doce*, primeiramente deve ser estabelecido um conjunto adequado de proposições atômicas e, em função dessas proposições, exprimir os enunciados fornecidos — premissas e conclusão. Sejam portanto os seguintes enunciados atômicos:

p: *Maria come doce*
q: *Maria engorda*
r: *Maria faz regime*

As premissas do exemplo em questão, expressas em função dos enunciados atômicos são, respectivamente:

$$p \rightarrow q,$$

$$q \rightarrow r,$$

$$\neg r$$

e a conclusão que queremos verificar:

$$p$$

O enunciado *Maria come doce*, simbolizado pela proposição atômica p, será verdade se puder ser provado a partir das premissas 1. 2. e 3.; será falso se for provado $\neg p$.

Como se trata de um problema pequeno, pode-se perfeitamente examinar exaustivamente todas as possíveis atribuições de valores-verdade às proposições p, q e r, para verificar a validade das possíveis conclusões ( p ou $\neg$ p). Todas as possíveis combinações estão na tabela que se segue:

| | | | | premissas | | | conclusões | | |
|---|---|---|---|---|---|---|---|---|---|
| | p | q | r | $p \rightarrow q$ | $q \rightarrow r$ | $\neg r$ | p | $\neg p$ | |
| $I_1$ | v | v | v | v | v | f | v | f | |
| $I_2$ | v | v | f | v | f | v | v | f | |
| $I_3$ | v | f | v | f | v | f | v | f | |
| $I_4$ | v | f | f | f | v | v | v | f | |
| $I_5$ | f | v | v | v | v | f | f | v | |
| $I_6$ | f | v | f | v | f | v | f | v | |
| $I_7$ | f | f | v | v | v | f | f | v | |
| $I_8$ | f | f | f | v | v | v | f | v | ← |

Para verificar a validade da conclusão p, as linhas nas quais todas as premissas são simultaneamente **v** são examinadas. A conclusão deve ser **v** também nessas linhas.

No exemplo, existe apenas uma linha onde todas as premissas são **v** — a última. Nesta linha a conclusão p é falsa, ou seja, *Maria come doce* é falso.

Pode ser verificado, entretanto, que o valor de $\neg p$ é verdade nesta linha. Portanto, dadas as premissas 1., 2. e 3., a conclusão verdadeira a que se chega a partir delas é $\neg p$, ou seja, *Maria* não *come doce*.

É sempre possível, através da tabela-verdade, determinar se uma conclusão segue das premissas num número finito de passos. Entretanto, a construção de tabelas-verdade pode facilmente se transformar num trabalho imenso, mesmo para um

computador, se existirem muitos enunciados atômicos. Como é sabido, com $n$ de tais enunciados, uma tabela-verdade requer $2^n$ linhas. Além disso, a tabela-verdade deverá ter, no mínimo, tantas colunas quantas forem as premissas, além da coluna da conclusão. Se a opção for por *forward chaining* ou *backward chaining*, não serão necessárias todas as linhas, mas de qualquer forma, um tempo razoável de computação pode ser consumido.

Como consequência, outros métodos, como por exemplo os métodos sintáticos, foram desenvolvidos para provar conclusões a partir de premissas.

# 4 Método Sintático de Prova de Teorema

Os métodos sintáticos de prova utilizam uma sequência lógica de tautologias, bem como teoremas provados previamente, para demonstrar que uma conclusão é uma consequência lógica de um dado conjunto de premissas. Uma demonstração deste tipo é chamada de *derivação.*

Na derivação, para provar sintaticamente o teorema:

$$p_1, p_2, \ldots, p_n \Rightarrow c$$

pode-se substituir qualquer das sentenças $p_i$, incluindo a conclusão c, por outra sentença que lhe seja logicamente equivalente; pode-se introduzir nas premissas qualquer sentença que seja logicamente implicada por qualquer coleção de sentenças participantes das premissas (até aquele presente estágio de derivação), e pode-se substituir a conclusão por qualquer sentença que logicamente a implique.

A derivação estará completa e o teorema estará provado quando (e se) ambos os lados de $\Rightarrow$ consistem de exatamente a mesma expressão. A existência de uma ocorrência deste tipo significa que a verdade de todas as premissas implica na verdade da conclusão, o que estabelece o teorema.

Na prova sintática de equivalência lógica, utiliza-se o fato de premissas e conclusão serem substituídas apenas por sentenças que lhes sejam logicamente equivalentes. Este fato garante a equivalência lógica entre o teorema original e a expressão a que se chega após substituições.

Deve ser notado que se em algum passo do processo de derivação for obtido um teorema conhecido, a derivação estará completa, uma vez que a presença de tal teorema garante a implicação lógica. Em tal passo as premissas implicam a

conclusão e, portanto, podem ser substituídas por ela, o que resulta na mesma expressão aparecendo em ambos os lados do símbolo $\Rightarrow$.

Os métodos sintáticos de prova de teoremas são mais diretos que os semânticos e usam apenas a estrutura do silogismo. Há vários algoritmos para prova automática de teoremas no Cálculo Proposicional por métodos sintáticos. Um deles é o Algoritmo de Wang, objeto deste trabalho, apresentado a seguir.

# 5 Algoritmo de Wang

O Algoritmo de Wang [Wang 60],[Wang 64] se aplica a expressões da forma:

$$p_1,p_2,p_3,\ldots,p_n \Rightarrow c_1,c_2,c_3,\ldots,c_n$$

onde os $p_i$ constituem as premissas e os $c_i$, constituem a conclusão:

$$c_1 \vee c_2 \vee c_3 \ldots \vee c_n$$

Em termos de valor-verdade — abordagem semântica — temos um teorema se o valor-verdade de:

$$p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow c_1 \vee c_2 \vee \ldots \vee c_n$$

for verdade. Embora no método automático de prova de teorema de Wang os valores-verdade não sejam utilizados, uma vez que se trata de uma abordagem sintática, uma compreensão semântica se faz necessária para um completo entendimento do algoritmo.

Para começar a provar um teorema usando o algoritmo de Wang, todas as premissas são escritas à esquerda do símbolo $\Rightarrow$, e a conclusão a que se deseja chegar é escrita à direita. Para o exemplo considerado na seção 3, teríamos:

$$p \rightarrow q,\ q \rightarrow r,\ \neg r \Rightarrow \neg p$$

Aplicamos transformações ao teorema, de forma a quebrá-lo em outros mais simples que, por sua vez, serão submetidos ao mesmo processo.

O algoritmo de Wang sempre termina em um número finito de passos, provando

$$p \Rightarrow q \text{ ou } p \not\Rightarrow q$$

O Algoritmo de Wang consiste de procedimentos — regras de transformação — que são aplicados recursivamente e de condições de parada.

A seguir, são descritas as 6 regras de transformação, $R_1$ a $R_6$ e as regras de término $R_7$ e $R_8$, que compõem o algoritmo.

## 5.1 Regras de Transformação

O objetivo dos procedimentos recursivos — regras de transformação — é o de remover conectivos de maneira que as regras de término possam ser aplicadas.

$R_1$ se uma das fórmulas tem a forma $\neg X$, podemos tirar a negação e move-la para o outro lado do símbolo $\Rightarrow$. X pode ser qualquer fórmula. Por exemplo:

$$p \vee q, \neg(r \wedge s), p \vee r \Rightarrow s,q$$

torna-se

$$p \vee q, p \vee r \Rightarrow s,q,r \wedge s$$

$$p,q,r \Rightarrow \neg s,t$$

torna-se

$$p,q,r,s \Rightarrow t$$

$R_2$ se uma fórmula à esquerda de $\Rightarrow$ tem a forma de $X \wedge Y$, ou se uma à direita tem a forma de $X \vee Y$, o conectivo pode ser substituído por uma vírgula. O $\wedge$ a ser removido deve ser o conectivo principal da sentença à esquerda de $\Rightarrow$. Observação análoga vale para o $\vee$ com relação à sentença à direita de $\Rightarrow$. Por exemplo:

$$p \wedge q, r \wedge(\neg p \vee s) \Rightarrow \neg p \wedge \neg r$$

torna-se

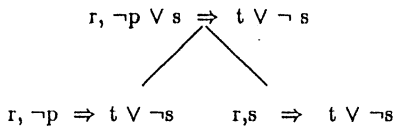
$$p,q,r,\neg p \vee s \Rightarrow \neg p \wedge \neg r$$

$$r \Rightarrow \neg s \vee \neg t$$

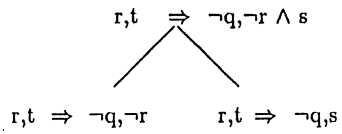
torna-se

$$r \Rightarrow \neg s,\neg t$$

$R_3$ se uma fórmula à esquerda de $\Rightarrow$ tem a forma $X \vee Y$, pode-se remover o operador $\vee$ e separar os seus dois argumentos, de forma a dividir o teorema original em dois novos teoremas. Cada um dos dois teoremas obtidos pela separação deve ser provado separadamente. O $\vee$ deve ser o principal conectivo à esquerda. Por exemplo:




$R_4$ se uma fórmula com a forma $X \wedge Y$ ocorre à direita de $\Rightarrow$, pode-se remover o operador $\wedge$ e separar os seus dois argumentos, de forma a dividir o teorema original em dois novos teoremas. Cada um dos dois teoremas obtidos pela separação deve ser provado separadamente. O $\wedge$ deve ser o principal conectivo à direita. Por exemplo:




É importante notar que a aplicação das regras $R_3$ ou $R_4$ provoca um crescimento exponencial do número de teoremas a serem provados. Em geral, se existirem um total de k $\wedge$ e $\vee$ como principais conectivos à esquerda e direita de $\Rightarrow$ respectivamente, um total de $2^k$ novos teoremas resultam como consequência da aplicação daquelas regras.

$R_5$ uma fórmula, em qualquer nível, que tenha a forma $(X \rightarrow Y)$, pode ser substituída pela fórmula equivalente $(\neg X \vee Y)$, eliminando-se com isso a implicação.

$R_6$ uma fórmula, em qualquer nível, que tenha a forma $(X \leftrightarrow Y)$, pode ser substituída pela fórmula equivalente $(X \rightarrow Y) \wedge (Y \rightarrow X)$, eliminando-se com isso a dupla implicação.



## 5.2 Regras de Parada

As regras anteriores devem ser aplicadas tantas vezes quantas forem necessárias, para a remoção dos conectivos $\leftrightarrow$, $\rightarrow$, $\neg$, $\wedge$ e $\vee$, até que:

$R_7$ um teorema é considerado provado, se alguma fórmula X ocorre em ambos os lados de $\Rightarrow$. Tal teorema é chamado de *axioma*. Nenhuma transformação será mais necessária neste teorema, muito embora possam existir outros a serem provados. O teorema original não estará provado até que todos os teoremas obtidos a partir dele tenham sido provados. Portanto, esta regra deve ser verificada para todo novo teorema que eventualmente resulte da aplicação das regras de transformação.

$R_8$ um teorema é provado inválido se todas as fórmulas que nele comparecem são símbolos de proposições individuais — isto é, não existem mais conectivos — e um mesmo símbolo não ocorre em ambos os lados de $\Rightarrow$. Se um teorema como este é encontrado, o algorítmo termina; a conclusão inicial não é uma consequência lógica das premissas

A seguir são apresentados os padrões das transformações propostas por Wang:

| | | | |
|---|---|---|---|
| $R_1$ | $(\ldots,\neg X,\ldots \Rightarrow \ldots, Z)$ | torna-se | $(\ldots \Rightarrow \ldots, Z,X)$ |
| $R_1$ | $(\ldots,Z \Rightarrow \ldots,\neg X,\ldots)$ | torna-se | $(\ldots, Z, X \Rightarrow \ldots)$ |
| $R_2$ | $(\ldots, X \wedge Y,\ldots \Rightarrow \ldots)$ | torna-se | $(\ldots, X, Y, \ldots \Rightarrow \ldots)$ |
| $R_2$ | $(\ldots \Rightarrow \ldots,X \vee Y,\ldots)$ | torna-se | $(\ldots \Rightarrow \ldots,X,Y,\ldots)$ |
| $R_3$ | $(\ldots, X \vee Y, \ldots \Rightarrow \ldots)$ | torna-se | $(\ldots,X,\ldots \Rightarrow \ldots)$ |
| | | e | $(\ldots,Y,\ldots \Rightarrow \ldots)$ |
| $R_4$ | $(\ldots \Rightarrow \ldots, X \wedge Y,\ldots)$ | torna-se | $(\ldots \Rightarrow \ldots, X, \ldots)$ |
| | | e | $(\ldots \Rightarrow \ldots, Y, \ldots)$ |
| $R_5$ | $(\ldots \Rightarrow \ldots, X \rightarrow Y, \ldots)$ | torna-se | $(\ldots, \ldots \Rightarrow \ldots, \neg X \vee Y, \ldots)$ |
| $R_5$ | $(\ldots, X \rightarrow Y, \ldots \Rightarrow \ldots)$ | torna-se | $(\ldots,\neg X \vee Y,\ldots \Rightarrow \ldots, \ldots)$ |
| $R_6$ | $(\ldots, X \leftrightarrow Y, \ldots \Rightarrow \ldots)$ | torna-se | $(\ldots,(X \rightarrow Y \wedge Y \rightarrow X), \ldots) \Rightarrow \ldots)$ |
| $R_6$ | $(\ldots \Rightarrow \ldots, X \leftrightarrow Y, \ldots)$ | torna-se | $(\ldots \Rightarrow \ldots, (X \rightarrow Y \wedge Y \rightarrow X), \ldots)$ |
| $R_7$ | $(\ldots,X,\ldots \Rightarrow \ldots,X,\ldots)$ | torna-se | true |

O algoritmo de Wang sempre converge para a solução de um dado problema. Toda aplicação de uma transformação conduz a algum progresso no sentido de

eliminar um conectivo e assim diminuir sintaticamente o tamanho do teorema — mesmo que com isso sejam criados outros teoremas, como é o caso da aplicação das regras $R_3$ e $R_4$.

A seguir, o algoritmo de Wang é aplicado na busca da conclusão para o exemplo da seção 3. As linhas foram rotuladas de maneira a faciliar referências a elas.

| Rótulo | | Comentários |
|---|---|---|
| S1: | p → q,q → r, ¬r ⇒ ¬p | teorema |
| S2: | ¬p ∨ q, ¬q ∨ r, ¬r ⇒ ¬p | duas aplicacões de $R_5$ |
| S3: | ¬p ∨ q,¬q ∨ r ⇒ ¬p,r | $R_1$ |
| S4: | ¬p,¬q ∨r ⇒ ¬p,r | S4 e S5 obtidos de S3 aplicando $R_3$.S4 é axioma |
| S5: | q,¬q ∨ r ⇒ ¬p,r | gerado de S3 com $R_3$ |
| S6: | q,¬q ⇒ ¬p,r | S6 e S7 gerados de S5 via $R_3$ |
| S7: | q,r ⇒ ¬p,r | axioma |
| S8: | q ⇒ ¬p,r,q | obtido de S6 usando $R_1$.Axioma |

O teorema original está provado uma vez que foi transformado em um conjunto de 3 axiomas e todos os teoremas intermediários foram também provados.

# 6 Implementações do Algoritmo de Wang

A seguir são mostradas tres implementações diferentes do algoritmo de Wang, na linguagem de programação lógica Prolog, bem como medidas de tempo de execução de cada implementação.

A primeira implementação é de autoria de L. Pereira e a segunda de I. Bratko. Ambas encontram-se em [Coelho 88].

A terceira implementação, bem mais eficiente, é de nossa autoria, e combina algumas das idéias usadas tanto na implementação de Pereira quanto na de Bratko.

Deve ser ressaltado que a implementação original de Bratko, apresentada na referência citada, tem algumas imprecisões que foram por nós corrigidas na versão apresentada neste trabalho.

Em todas as implementações, os conectivos lógicos são definidos pelos seguintes operadores:

```
:- op(690,xfy,=>).          % teorema
:- op(670,xfy,<->).         % equivalencia
:- op(650,xfy,->).          % implicacao
:- op(600,xfy,v).           % disjuncao
:- op(550,xfy,&).           % conjuncao
:- op(500,fy,n).            % negacao
```

A entrada para todos os programas tem dois possíveis formatos:

1. uma fórmula bem formada do CP, como por exemplo

   ```
   p <-> q -> (p & q) v (n p & n q).

   p & ( p -> q) -> q.
   ```

2. duas fórmulas bem formadas $f_1$ e $f_2$ do CP, onde se queira provar que

   $$f_1 \Rightarrow f_2$$

   Por exemplo

   ```
   p <-> q => (p & q) v (n p & n q).

   p & ( p -> q) => q.
   ```

Observe que qualquer dos formatos pode ser utilizado para entrar o mesmo teorema.

Todos os programas são ativados pelo predicado `teorema/2`, onde o primeiro argumento é o teorema a ser provado e o segundo argumento é uma variável, cujo valor será `v` se o primeiro argumento for um teorema, `f` caso contrário.

Por exemplo:

```
?- teorema( p <-> q => (p & q) v (n p & n q), Val).
```

será bem sucedido com `Val` unificado com `v`, enquanto que

```
?- teorema( p & q -> p v r <-> p v p, Val).
```

será bem sucedido com `Val` unificado com `f`.

## 6.1 Implementação de L. Pereira

Nesta implementação é utilizado um operador adicional : usado para delimitar as listas que fazem parte da estrutura utilizada. Ele é definido por:

```
:- op(550,xfy,:).
```

A idéia geral usada na versão de Pereira é a de utilizar uma estrutura da forma:

```
[] : []  =>  [] : []
1    2       3    4
```

Se a entrada T do usuário for fornecida sem o símbolo `=>` , a prova é iniciada na estrutura:

```
[] : []  =>  [] : [T]
```

enquanto que se a entrada for do tipo

```
L  =>  R
```

a prova é iniciada na estrutura:

```
[] : [L]  =>  [] : [R]
```

A partir daí, a implementação tenta aplicar as regras $R_5$, $R_6$ e $R_1$, nesta ordem, a fim de eliminar os conectivos `->` , `<->` , e `n` (que representa a negação). A seguir, tenta aplicar a regra $R_2$, que substitui nas listas L e R respectivamente, os conectivos $\wedge$ e $\vee$ por vírgulas, a fim de transformar uma fórmula do tipo:

$$p_1 \wedge p_2 \Rightarrow c_1 \vee c_2$$

para a notação:

$$p_1,p_2 \Rightarrow c_1,c_2$$

É importante notar que até este ponto as listas 1 e 3 não foram utilizadas.

Quando não for possível a aplicação das regras $R_1$,$R_2$, $R_5$ e/ou $R_6$, o programa tenta aplicar as regras $R_3$ ou $R_4$ que dividem o teorema que está sendo provado em dois outros teoremas, os quais por sua vez devem ser provados independentemente.

Quando não for possível aplicar nenhuma das regras anteriores, é porque a cabeça das listas 2 ou 4 é um átomo e, como tal, ele é transferido para a lista 1 ou 3 respectivamente. A função das listas 1 e 3 é a de acumular os átomos contidos no teorema original, de maneira a permitir a verificação de uma tautologia quando se obtém a estrutura

$$Lf : [\,] \Rightarrow Rf : [\,]$$

verificando simplesmente se um átomo da lista Lf está também presente na lista Rf, caso contrário o teorema não é válido.

Segue-se uma listagem do programa.

### 6.1.1 Listagem do Programa

```
teorema( L => R,v) :-
    prove([]:[L] => []:[R]),
    !.

teorema( L => R,f) :-
    !.

teorema(T,v) :-
    prove([]:[]=>[]:[T]),
    !.

teorema(T,f).


prove(E1) :-
    regra(E1,E2,Reg),
    !,
    prove(E2).
```

```
% caso de v no lado esquerdo
prove(L : [H v I|T] => R) :-
    !,
    Esq = L : [H|T] => R,
    Dir = L : [I|T] => R,
    prove(Esq),
    prove(Dir).

% caso de & no lado direito
prove(L => R : [H & I|T]) :-
    !,
    Esq = L => R : [H|T],
    Dir = L => R : [I|T],
    prove(Esq),
    prove(Dir).

% caso de atomo
prove(L : [H|T] => R) :-
    !,
    prove([H|L] : T => R).

prove(L => R : [H|T]) :-
    !,
    prove(L => [H|R] : T).

% verifica se e tautologia
prove(T) :-
    tautologia(T).


% casos onde -> aparece em um dos lados
regra(L : [H -> I|T] => R, L : [n H v I|T] => R, regra_5).

regra(L => R : [H -> I|T], L => R : [n H v I|T], regra_5).

% caso onde <-> aparece em um dos lados
regra(L : [H <-> I|T] => R, L : [(H -> I) & (I -> H)|T] => R,
                                                    regra_6).

regra(L => R : [H <-> I|T], L => R : [(H -> I) & (I -> H)|T],
                                                    regra_6).
```

```
% casos onde n (negacao) aparece em um dos lados
regra(L : [n H|T] => R : R2, L : T => R : [H|R2],regra_1).

regra(L1 : L2 => R : [n H|T], L1 : [H|L2] => R : T ,regra_1).

% caso do conectivo & do lado esquerdo de =>
regra(L : [H & I|T] => R, L : [H,I|T] => R,regra_2).

% caso do conectivo v do lado direito de =>
regra(L => R : [H v I | T], L => R : [H,I|T],regra_2).


tautologia(L : [] => R : []) :-
    pertence(M,L),
    pertence(M,R),
    !.


pertence(H,[H|_]).
pertence(I,[_|T]) :-
    pertence(I,T).
```

### 6.1.2 Exemplos de Execução

A fim de exemplificar a execução deste programa, serão acrescentados a ele predicados de leitura e escrita. Os novos predicados, bem como aqueles que sofreram algumas modicações devido a esse acréscimo, são listados a seguir.

```
pereira :-
    info_intro('Pereira'),
    repeat,
    entrada(Formula),
    (Formula = fim ;
     teorema(Formula,Valor), info_valor(Valor), fail).
```

```
prove(E1) :-
    regra(E1,E2,Reg),
    !,
    info_regra(E2,Reg),
    prove(E2).

% caso de v no lado esquerdo
prove(L : [H v I|T] => R) :- !,
    Esq = L : [H|T] => R,
    Dir = L : [I|T] => R,
    info_regra(Esq,Dir,regra_3),
    info_provando(Esq),
    prove(Esq),
    info_provando(Dir),
    prove(Dir).

% caso de & no lado direito
prove(L => R : [H & I|T]) :- !,
    Esq = L => R : [H|T],
    Dir = L => R : [I|T],
    info_regra(Esq,Dir,regra_4),
    info_provando(Esq),
    prove(Esq),
    info_provando(Dir),
    prove(Dir).

% caso de atomo
prove(L : [H|T] => R) :- !,prove([H|L] : T => R).

prove(L => R : [H|T]) :- !,prove(L => [H|R] : T).

% verifica se e tautologia
prove(T) :-
    tautologia(T),
    !,
    info_tauto(T).

prove(_) :-
     info_falha,
     fail.
```

```
% Predicados auxiliares de leitura e escrita que serao usados
% para mostrar a execucao das tres implementacoes do algoritmo
info_intro(P):-
    nl,tab(4),
    write('Provador de Teoremas do Calculo Proposicional'),
    nl,tab(4),
    write('Algoritmo de Wang - Implementacao de '),
    write(P),
    nl,nl.


entrada(F) :-
    nl,tab(4),
    write('Entre com a formula (fim para terminar):'),
    nl,nl,read(F).


info_valor(v):-
    nl,tab(4),
    write('** A formula e um teorema **'),
    nl.

info_valor(f) :-
    nl,nl,tab(4),
    write('** A formula nao e um teorema **'),
    nl.


info_regra(E,Reg) :-
    nl,write(Reg),tab(3),
    write(E).

info_regra(E,D,Reg) :-
    nl,nl,
    write('Para provar formula anterior, pela regra '),
    write(Reg),
    write(' devemos provar:'),nl,nl,
    tab(15),write(E),nl,
    tab(30),write(e),nl,
    tab(15),write(D),nl,nl.
```

```
info_provando(E) :-
    write('Provando  '),
    write(E).


info_tauto(T) :-
    nl,tab(10),
    write(T),
    write('    Ramo provado'),nl,nl.


info_falha :-
    write('    Ramo nao pode ser provado').
```

Segue-se um exemplo de execução do programa `pereira/0`

```
?-pereira.

    Provador de Teoremas do Calculo Proposicional
    Algoritmo de Wang - Implementacao de Pereira


    Entre com a formula (fim para terminar):

p & q -> r v q & p => p -> q -> r v q & p.

regra_5   [] : [n (p & q) v r v q & p] => [] : [p -> q -> r v q & p]
regra_5   [] : [n (p & q) v r v q & p] => [] : [n p v (q -> r v q & p)]
regra_2   [] : [n (p & q) v r v q & p] => [] : [n p,q -> r v q & p]
regra_1   [] : [p,n (p & q) v r v q & p] => [] : [q -> r v q & p]
regra_5   [] : [p,n (p & q) v r v q & p] => [] : [n q v r v q & p]
regra_2   [] : [p,n (p & q) v r v q & p] => [] : [n q,r v q & p]
regra_1   [] : [q,p,n (p & q) v r v q & p] => [] : [r v q & p]
regra_2   [] : [q,p,n (p & q) v r v q & p] => [] : [r,q & p]
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
[p,q] : [n (p & q)] => [] : [r,q & p]
```

e

```
                [p,q] : [r v q & p] => [] : [r,q & p]

Provando  [p,q] : [n (p & q)] => [] : [r,q & p]
regra_1   [p,q] : [] => [] : [p & q,r,q & p]
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
                [p,q] : [] => [] : [p,r,q & p]
                                e
                [p,q] : [] => [] : [q,r,q & p]
```

```
Provando  [p,q] : [] => [] : [p,r,q & p]
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
                [p,q] : [] => [r,p] : [q]
                                e
                [p,q] : [] => [r,p] : [p]
```

```
Provando  [p,q] : [] => [r,p] : [q]
          [p,q] : [] => [q,r,p] : []     Ramo provado

Provando  [p,q] : [] => [r,p] : [p]
          [p,q] : [] => [p,r,p] : []     Ramo provado

Provando  [p,q] : [] => [] : [q,r,q & p]
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
                [p,q] : [] => [r,q] : [q]
                                e
                [p,q] : [] => [r,q] : [p]
```

```
Provando  [p,q] : [] => [r,q] : [q]
          [p,q] : [] => [q,r,q] : []     Ramo provado

Provando  [p,q] : [] => [r,q] : [p]
          [p,q] : [] => [p,r,q] : []     Ramo provado

Provando  [p,q] : [r v q & p] => [] : [r,q & p]
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
[p,q] : [r] => [] : [r,q & p]
                e
[p,q] : [q & p] => [] : [r,q & p]
```

Provando `[p,q] : [r] => [] : [r,q & p]`

Para provar formula anterior, pela regra regra_4 devemos provar:

```
[r,p,q] : [] => [r] : [q]
                e
[r,p,q] : [] => [r] : [p]
```

Provando `[r,p,q] : [] => [r] : [q]`
`[r,p,q] : [] => [q,r] : []` Ramo provado

Provando `[r,p,q] : [] => [r] : [p]`
`[r,p,q] : [] => [p,r] : []` Ramo provado

Provando `[p,q] : [q & p] => [] : [r,q & p]`
regra_2 `[p,q] : [q,p] => [] : [r,q & p]`

Para provar formula anterior, pela regra regra_4 devemos provar:

```
[p,q,p,q] : [] => [r] : [q]
                  e
[p,q,p,q] : [] => [r] : [p]
```

Provando `[p,q,p,q] : [] => [r] : [q]`
`[p,q,p,q] : [] => [q,r] : []` Ramo provado

Provando `[p,q,p,q] : [] => [r] : [p]`
`[p,q,p,q] : [] => [p,r] : []` Ramo provado

** A formula e um teorema **

Entre com a formula (fim para terminar):

```
p v q -> p v r => p v (p -> r).
```

```
regra_5   [] : [n (p v q) v p v r] => [] : [p v (p -> r)]
regra_2   [] : [n (p v q) v p v r] => [] : [p,p -> r]
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
                [] : [n (p v q)] => [] : [p,p -> r]
                                e
                [] : [p v r] => [] : [p,p -> r]
```

```
Provando  [] : [n (p v q)] => [] : [p,p -> r]
regra_1   [] : [] => [] : [p v q,p,p -> r]
regra_2   [] : [] => [] : [p,q,p,p -> r]
regra_5   [] : [] => [p,q,p] : [n p v r]
regra_2   [] : [] => [p,q,p] : [n p,r]
regra_1   [] : [p] => [p,q,p] : [r]
          [p] : [] => [r,p,q,p] : []    Ramo provado
```

```
Provando  [] : [p v r] => [] : [p,p -> r]
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
                [] : [p] => [] : [p,p -> r]
                                e
                [] : [r] => [] : [p,p -> r]
```

```
Provando  [] : [p] => [] : [p,p -> r]
regra_5   [p] : [] => [p] : [n p v r]
regra_2   [p] : [] => [p] : [n p,r]
regra_1   [p] : [p] => [p] : [r]
          [p,p] : [] => [r,p] : []    Ramo provado
```

```
Provando  [] : [r] => [] : [p,p -> r]
regra_5   [r] : [] => [p] : [n p v r]
regra_2   [r] : [] => [p] : [n p,r]
regra_1   [r] : [p] => [p] : [r]
          [p,r] : [] => [r,p] : []    Ramo provado
```

** A formula e um teorema **

```
Entre com a formula (fim para terminar):

n p <-> n q <-> n r => p & r.
regra_6   [] : [(n p -> (n q <-> n r)) & ((n q <-> n r) -> n p)] =>
                                                        [] : [p & r]
regra_2   [] : [n p -> (n q <-> n r),(n q <-> n r) -> n p] =>
                                                        [] : [p & r]
regra_5   [] : [n n p v (n q <-> n r),(n q <-> n r) -> n p] =>
                                                        [] : [p & r]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [] : [n n p,(n q <-> n r) -> n p] => [] : [p & r]
                                    e
                [] : [n q <-> n r,(n q <-> n r) -> n p] => [] : [p & r]

Provando  [] : [n n p,(n q <-> n r) -> n p] => [] : [p & r]
regra_1   [] : [(n q <-> n r) -> n p] => [] : [n p,p & r]
regra_5   [] : [n (n q <-> n r) v n p] => [] : [n p,p & r]
regra_1   [] : [p,n (n q <-> n r) v n p] => [] : [p & r]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [] : [p,n (n q <-> n r) v n p] => [] : [p]
                                    e
                [] : [p,n (n q <-> n r) v n p] => [] : [r]

Provando  [] : [p,n (n q <-> n r) v n p] => [] : [p]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [p] : [n (n q <-> n r)] => [] : [p]
                                    e
                [p] : [n p] => [] : [p]

Provando  [p] : [n (n q <-> n r)] => [] : [p]
regra_1   [p] : [] => [] : [n q <-> n r,p]
regra_6   [p] : [] => [] : [(n q -> n r) & (n r -> n q),p]

Para provar formula anterior, pela regra regra_4 devemos provar:
```

```
                [p] : [] => [] : [n q -> n r,p]
                                  e
                [p] : [] => [] : [n r -> n q,p]

Provando  [p] : [] => [] : [n q -> n r,p]
regra_5   [p] : [] => [] : [n n q v n r,p]
regra_2   [p] : [] => [] : [n n q,n r,p]
regra_1   [p] : [n q] => [] : [n r,p]
regra_1   [p] : [] => [] : [q,n r,p]
regra_1   [p] : [r] => [q] : [p]
          [r,p] : [] => [p,q] : []    Ramo provado

Provando  [p] : [] => [] : [n r -> n q,p]
regra_5   [p] : [] => [] : [n n r v n q,p]
regra_2   [p] : [] => [] : [n n r,n q,p]
regra_1   [p] : [n r] => [] : [n q,p]
regra_1   [p] : [] => [] : [r,n q,p]
regra_1   [p] : [q] => [r] : [p]
          [q,p] : [] => [p,r] : []    Ramo provado

Provando  [p] : [n p] => [] : [p]
regra_1   [p] : [] => [] : [p,p]
          [p] : [] => [p,p] : []    Ramo provado

Provando  [] : [p,n (n q <-> n r) v n p] => [] : [r]
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
                [p] : [n (n q <-> n r)] => [] : [r]
                                  e
                [p] : [n p] => [] : [r]

Provando  [p] : [n (n q <-> n r)] => [] : [r]
regra_1   [p] : [] => [] : [n q <-> n r,r]
regra_6   [p] : [] => [] : [(n q -> n r) & (n r -> n q),r]
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
                [p] : [] => [] : [n q -> n r,r]
                                  e
                [p] : [] => [] : [n r -> n q,r]
```

```
Provando  [p] : [] => [] : [n q -> n r,r]
regra_5   [p] : [] => [] : [n n q v n r,r]
regra_2   [p] : [] => [] : [n n q,n r,r]
regra_1   [p] : [n q] => [] : [n r,r]
regra_1   [p] : [] => [] : [q,n r,r]
regra_1   [p] : [r] => [q] : [r]
          [r,p] : [] => [r,q] : []     Ramo provado

Provando  [p] : [] => [] : [n r -> n q,r]
regra_5   [p] : [] => [] : [n n r v n q,r]
regra_2   [p] : [] => [] : [n n r,n q,r]
regra_1   [p] : [n r] => [] : [n q,r]
regra_1   [p] : [] => [] : [r,n q,r]
regra_1   [p] : [q] => [r] : [r]     Ramo nao pode ser provado

    ** A formula nao e um teorema **

Entre com a formula (fim para terminar):

fim.
```

## 6.2 Implementação de Bratko

A idéia usada por Bratko é, antes de decompor o teorema, verificar pela existência de uma mesma fórmula, não necessariamente atômica como no caso anterior, que compareça em ambos os lados do símbolo `=>` . Isto pode ser visto mais claramente na seção 6.2.2, na execução do primeiro exemplo.

Procura-se, pois, por uma fórmula que faça parte, simultaneamente, de uma conjunção à esquerda e de uma disjunção à direita do símbolo `=>` .

Esta implementação não faz uso de outra estrutura que não a do próprio teorema. Os predicados fundamentais para um entendimento de como funciona o programa são `conconc/3` e `disconc/3`, ambos não determinísticos, definidos por:

```
conconc(true,Exp,Exp).

conconc(Term & Exp1,Exp2,Term & Exp3) :-
    conconc(Exp1,Exp2,Exp3).
```

```
disconc(false,Exp,Exp).

disconc(Term v Exp1,Exp2,Term v Exp3) :-
    disconc(Exp1,Exp2,Exp3).
```

A idéia que o predicado `conconc/3` implementa é a de extrair uma fórmula de uma conjunção de fórmulas, enquanto que `disconc/3` faz o mesmo em uma disjunção de fórmulas.

Seguem-se vários exemplos de execução de `conconc/3` quando ele é ativado com seu terceiro argumento unificado com a fórmula

```
(q <-> r) & (p v q) & n (q -> p) & m & n r
```

```
?- conconc(L1,(E1 v E2) & L2,L).

L1 = (q <-> r) & true
E1 = p
E2 = q
L2 = n (q -> p) & m & n r
L = (q <-> r) & (p v q) & n (q -> p) & m & n r  ->;
no

?- conconc(L1,n E1 & L2,L).

L1 = (q <-> r) & (p v q) & true
E1 = q -> p
L2 = m & n r
L = (q <-> r) & (p v q) & n (q -> p) & m & n r  ->;
no

?- conconc(L1,E1 & L2,L).

L1 = true
E1 = q <-> r
L2 = (p v q) & n (q -> p) & m & n r
L = (q <-> r) & (p v q) & n (q -> p) & m & n r  ->;


L1 = (q <-> r) & true
```

```
E1 = p v q
L2 = n (q -> p) & m & n r
L = (q <-> r) & (p v q) & n (q -> p) & m & n r  ->;


L1 = (q <-> r) & (p v q) & true
E1 = n (q -> p)
L2 = m & n r
L = (q <-> r) & (p v q) & n (q -> p) & m & n r  ->;


L1 = (q <-> r) & (p v q) & n (q -> p) & true
E1 = m
L2 = n r
L = (q <-> r) & (p v q) & n (q -> p) & m & n r  ->;
no
```

A seguir são mostrados vários exemplos de execução de `disconc/3` quando ele é ativado com seu terceiro argumento unificado com a fórmula

```
(q <-> r) v p & q v n (q -> p) v m v n r
```

```
?- disconc(L1,(E1 & E2) v L2,L).

L1 = (q <-> r) v false
E1 = p
E2 = q
L2 = n (q -> p) v m v n r
L = (q <-> r) v p & q v n (q -> p) v m v n r  ->;
no

?- disconc(L1,n E1 v L2,L).

L1 = (q <-> r) v p & q v false
E1 = q -> p
L2 = m v n r
L = (q <-> r) v p & q v n (q -> p) v m v n r  ->;
no

?- disconc(L1,E1 v L2,L).
```

```
L1 = false
E1 = q <-> r
L2 = p & q v n (q -> p) v m v n r
L = (q <-> r) v p & q v n (q -> p) v m v n r  ->;


L1 = (q <-> r) v false
E1 = p & q
L2 = n (q -> p) v m v n r
L = (q <-> r) v p & q v n (q -> p) v m v n r  ->;


L1 = (q <-> r) v p & q v false
E1 = n (q -> p)
L2 = m v n r
L = (q <-> r) v p & q v n (q -> p) v m v n r  ->;


L1 = (q <-> r) v p & q v n (q -> p) v false
E1 = m
L2 = n r
L = (q <-> r) v p & q v n (q -> p) v m v n r  ->;
no
```

Segue-se uma listagem do programa de Bratko, onde, como já foi dito anteriormente, foram realizadas algumas correções.

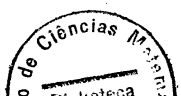
### 6.2.1 Listagem do Programa

```
teorema(L => R,v) :-
    prove(L&true => R v false),
    !.

teorema(L => R,f) :-
    !.

teorema(T,Val) :-
    teorema(true => T,Val).
```

```
prove(T) :-
    tautologia(T),
    !.

prove(L => R) :-
    conconc(L1,(E1 v E2)&L2,L),
    conconc(L1,L2,LL),
    !,
    Esq = E1 & LL => R,
    Dir = E2 & LL => R,
    prove(Esq),
    prove(Dir).

prove(L => R) :-
    disconc(R1,(E1&E2) v R2,R),
    disconc(R1,R2,RR),
    !,
    Esq = L => E1 v RR,
    Dir = L => E2 v RR,
    prove(Esq),
    prove(Dir).

prove(T) :-
    regra(T,T1,Reg),
    !,
    prove(T1).


tautologia(L => R) :-
    conmember(Exp,L),
    dismember(Exp,R),
    !.


% caso onde n (negacao) aparece em um dos lados
regra(L => R, LL =>  Exp v R,regra_1) :-
    conconc(L1,n Exp&L2,L),
    conconc(L1,L2,LL),
    !.
```

```
regra(L => R, Exp & L => RR,regra_1) :-
    disconc(R1,n Exp v R2,R),
    disconc(R1,R2,RR),
    !.

% caso do conectivo & e do lado esquerdo de =>
regra(L => R, LL => R,regra_2) :-
    conconc(L1,(A & B) & L2,L),
    conconc(L1,A & (B & L2),LL),
    !.

% caso do conectivo v do lado direito de =>
regra(L => R, L => RR,regra_2) :-
    disconc(R1,(A v B) v R2,R),
    disconc(R1,A v (B v R2),RR),
    !.

% caso onde <-> ou -> aparece em um dos lados de =>
regra(L => R, LL => R,regra_5/6) :-
    conconc(L1,Exp&L2,L),
    rule(Exp,Exp1),
    conconc(L1,Exp1&L2,LL),
    !.

regra(L => R,L => RR,regra_5/6) :-
    disconc(R1,Exp v R2,R),
    rule(Exp,Exp1),
    !,
    disconc(R1,Exp1 v R2, RR),
    !.

% regras para eliminar os conetivos -> e <->
rule(A -> B, n A v B).

rule(A <-> B, (n A & n B) v (A & B)).


conconc(true,Exp,Exp).

conconc(Term&Exp1,Exp2,Term&Exp3) :-
    conconc(Exp1,Exp2,Exp3).
```





```
conmember(Term,Exp) :-
    conconc(Exp1,Term&Exp2,Exp).


disconc(false,Exp,Exp).

disconc(Term v Exp1,Exp2,Term v Exp3) :-
    disconc(Exp1,Exp2,Exp3).


dismember(Term,Exp) :-
    disconc(Exp1,Term v Exp2,Exp).
```

Analogamente à seção 6.1.2, serão acrescentados ao programa anterior os mesmos predicados de leitura e escrita, de maneira que sua execução possa ser exibida. Também serão apresentados apenas aqueles predicados que foram alterados com tal acréscimo.

### 6.2.2 Exemplos de Execução

```
bratko :-
    info_intro('Bratko'),
    repeat,
    entrada(Formula),
    (Formula = fim ;
     teorema(Formula,Valor), info_valor(Valor), fail).


prove(T) :-
    tautologia(T),!.

prove(L => R) :-
     conconc(L1,(E1 v E2)&L2,L),
     conconc(L1,L2,LL),!,
     Esq = E1 & LL => R,
     Dir = E2 & LL => R,
     info_regra(Esq,Dir,regra_3),
     info_provando(Esq),
```

```
        prove(Esq),
        info_provando(Dir),
        prove(Dir).

prove(L => R) :-
        disconc(R1,(E1&E2) v R2,R),
        disconc(R1,R2,RR),!,
        Esq = L => E1 v RR,
        Dir = L => E2 v RR,
        info_regra(Esq,Dir,regra_4),
        info_provando(Esq),
        prove(Esq),
        info_provando(Dir),
        prove(Dir).

prove(T) :-
        regra(T,T1,Reg),!,
        info_regra(T1,Reg),
        prove(T1).

prove(_) :-
        info_falha,!,fail.

tautologia(L => R) :-
        conmember(Exp,L),
        dismember(Exp,R),
        !,
        info_tauto(L => R).
```

É mostrado a seguir a execução de bratko/0 com os mesmos dados de entrada usados na execução de pereira/0

```
?- bratko.

    Provador de Teoremas do Calculo Proposicional
    Algoritmo de Wang - Implementacao de Bratko


    Entre com a formula (fim para terminar):

p & q -> r v q & p => p -> q -> r v q & p.
```

```
regra_5 / 6   (n (p & q) v r v q & p) & true =>
                                  (p -> q -> r v q & p) v false
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
              n (p & q) & true => (p -> q -> r v q & p) v false
                              e
              (r v q & p) & true => (p -> q -> r v q & p) v false
```

```
Provando  n (p & q) & true => (p -> q -> r v q & p) v false
regra_1   true => p & q v (p -> q -> r v q & p) v false
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
              true => p v (p -> q -> r v q & p) v false
                              e
              true => q v (p -> q -> r v q & p) v false
```

```
Provando  true => p v (p -> q -> r v q & p) v false
regra_5 / 6   true => p v (n p v (q -> r v q & p)) v false
regra_2   true => p v n p v (q -> r v q & p) v false
regra_1   p & true => p v (q -> r v q & p) v false
          p & true => p v (q -> r v q & p) v false    Ramo provado
```

```
Provando  true => q v (p -> q -> r v q & p) v false
regra_5 / 6   true => q v (n p v (q -> r v q & p)) v false
regra_2   true => q v n p v (q -> r v q & p) v false
regra_1   p & true => q v (q -> r v q & p) v false
regra_5 / 6   p & true => q v (n q v r v q & p) v false
regra_2   p & true => q v n q v (r v q & p) v false
regra_1   q & p & true => q v (r v q & p) v false
          q & p & true => q v (r v q & p) v false    Ramo provado
```

```
Provando  (r v q & p) & true => (p -> q -> r v q & p) v false
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
              r & true => (p -> q -> r v q & p) v false
                              e
              (q & p) & true => (p -> q -> r v q & p) v false
```

```
Provando  r & true => (p -> q -> r v q & p) v false
regra_5 / 6   r & true => (n p v (q -> r v q & p)) v false
regra_2   r & true => n p v (q -> r v q & p) v false
regra_1   p & r & true => (q -> r v q & p) v false
regra_5 / 6   p & r & true => (n q v r v q & p) v false
regra_2   p & r & true => n q v (r v q & p) v false
regra_1   q & p & r & true => (r v q & p) v false
regra_2   q & p & r & true => r v q & p v false
          q & p & r & true => r v q & p v false    Ramo provado

Provando  (q & p) & true => (p -> q -> r v q & p) v false
regra_2   q & p & true => (p -> q -> r v q & p) v false
regra_5 / 6   q & p & true => (n p v (q -> r v q & p)) v false
regra_2   q & p & true => n p v (q -> r v q & p) v false
regra_1   p & q & p & true => (q -> r v q & p) v false
regra_5 / 6   p & q & p & true => (n q v r v q & p) v false
regra_2   p & q & p & true => n q v (r v q & p) v false
regra_1   q & p & q & p & true => (r v q & p) v false
regra_2   q & p & q & p & true => r v q & p v false
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
q & p & q & p & true => q v r v false
                  e
q & p & q & p & true => p v r v false
```

```
Provando  q & p & q & p & true => q v r v false
          q & p & q & p & true => q v r v false    Ramo provado

Provando  q & p & q & p & true => p v r v false
          q & p & q & p & true => p v r v false    Ramo provado
```

** A formula e um teorema **

Entre com a formula (fim para terminar):

```
p v q -> p v r => p v (p -> r).

regra_2   (p v q -> p v r) & true => p v (p -> r) v false
```

```
regra_5 / 6   (n (p v q) v p v r) & true => p v (p -> r) v false
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
              n (p v q) & true => p v (p -> r) v false
                                  e
              (p v r) & true => p v (p -> r) v false
```

```
Provando  n (p v q) & true => p v (p -> r) v false
regra_1   true => (p v q) v p v (p -> r) v false
regra_2   true => p v q v p v (p -> r) v false
regra_5 / 6   true => p v q v p v (n p v r) v false
regra_2   true => p v q v p v n p v r v false
regra_1   p & true => p v q v p v r v false
          p & true => p v q v p v r v false    Ramo provado
```

```
Provando  (p v r) & true => p v (p -> r) v false
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
              p & true => p v (p -> r) v false
                                  e
              r & true => p v (p -> r) v false
```

```
Provando  p & true => p v (p -> r) v false
          p & true => p v (p -> r) v false    Ramo provado

Provando  r & true => p v (p -> r) v false
regra_5 / 6   r & true => p v (n p v r) v false
regra_2   r & true => p v n p v r v false
          r & true => p v n p v r v false    Ramo provado

    ** A formula e um teorema **


Entre com a formula (fim para terminar):

n p <-> n q <-> n r => p & r.
```

Para provar formula anterior, pela regra regra_4 devemos provar:

```
                (n p <-> n q <-> n r) & true => p v false
                                  e
                (n p <-> n q <-> n r) & true => r v false

Provando  (n p <-> n q <-> n r) & true => p v false
regra_5 / 6   (n n p & n (n q <-> n r) v n p & (n q <-> n r)) & true =>
                                                          p v false
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
                (n n p & n (n q <-> n r)) & true => p v false
                                  e
                (n p & (n q <-> n r)) & true => p v false

Provando  (n n p & n (n q <-> n r)) & true => p v false
regra_2   n n p & n (n q <-> n r) & true => p v false
regra_1   n (n q <-> n r) & true => n p v p v false
regra_1   true => (n q <-> n r) v n p v p v false
regra_1   p & true => (n q <-> n r) v p v false
          p & true => (n q <-> n r) v p v false    Ramo provado

Provando  (n p & (n q <-> n r)) & true => p v false
regra_2   n p & (n q <-> n r) & true => p v false
regra_1   (n q <-> n r) & true => p v p v false
regra_5 / 6   (n n q & n n r v n q & n r) & true => p v p v false
```

Para provar formula anterior, pela regra regra_3 devemos provar:

```
                (n n q & n n r) & true => p v p v false
                                  e
                (n q & n r) & true => p v p v false

Provando  (n n q & n n r) & true => p v p v false
regra_2   n n q & n n r & true => p v p v false
regra_1   n n r & true => n q v p v p v false
regra_1   true => n r v n q v p v p v false
regra_1   r & true => n q v p v p v false
regra_1   q & r & true => p v p v false    Ramo nao pode ser provado
```

** A formula nao e um teorema **

```
Entre com a formula (fim para terminar):

fim.
```

## 6.3 Implementação de MCM-MCN

Apresentamos a seguir a implementação por nós proposta do Algoritmo de Wang. Com o objetivo de favorecer a eficiência, procuramos, nesta implementação, reunir os aspectos positivos das duas implementações apresentadas anteriormente.

Embora a implementação de Bratko seja mais elegante que a de Pereira, ela é, na grande maioria dos casos, menos eficiente (ver seção 7). Isto se deve, em parte, à estrutura de dados utilizada.

Deve ser observado que a estrutura de dados lista, usada por Pereira na sua implementação, melhora sensivelmente o tempo de execução do algoritmo. Por esta razão, decidimos usar esta mesma estrutura de dados.

Apesar da implementação de Bratko ser mais ineficiente, ela é vantajosa no sentido de permitir a detecção imediata da presença de uma mesma fórmula em ambos os lados do símbolo `=>` , sem necessidade de analizar seus componentes atômicos, como acontece na implementação de Pereira.

Uma das alterações por nós proposta é a de realizar a transformação da fórmula de entrada, logo no início, de maneira a simplificar negações bem como remover implicações `->` e equivalências `<->` .

Este passo inicial melhora o tempo de execução pois diminui o número de movimentos de uma mesma fórmula de um lado para outro do símbolo `=>` , a fim de simplificar as negações.

Uma outra melhora acrescentada refere-se à verificação de tautologia. Deve ser observado que Pereira somente verifica pela existência de uma tautologia quando a estrutura é da forma

$$\underset{1}{[L]} : \underset{2}{[\,]} \Rightarrow \underset{3}{[R]} : \underset{4}{[\,]}$$

onde L e R são listas contendo apenas átomos.

Entretanto, não há necessidade das listas 2 e 4 estarem vazias para realizar esta

verificação. Esta última abordagem foi por nos utilizada e, em parte, é devido a ela a melhora no tempo de execução da implementação por nós proposta.

### 6.3.1 Listagem do Programa

```
teorema( L => R,v) :-
     transforme(L,L1),
     transforme(R,R1),
     prove([]:[L1] => []:[R1]),
     !.

teorema( L => R,f) :- !.

teorema(T,v) :-
    transforme(T,T1),
    prove([]:[]=>[]:[T1]),
    !.

teorema(T,f).


prove(E1) :-
    regra(E1,E2,Reg),
    !,
    prove(E2).

% caso de v no lado esquerdo
prove(L : [H v I|T] => R) :- !,
    Esq = L : [H|T] => R,
    Dir = L : [I|T] => R,
    prove(Esq),
    prove(Dir).

% caso de & no lado direito
prove(L => R : [H & I|T]) :- !,
    Esq = L => R : [H|T],
    Dir = L => R : [I|T],
    prove(Esq),
    prove(Dir).
```

```
% caso de atomo
prove(L : [H|T] => R1 : R2) :-
    pertence(H,R1),
    !.

prove(L1 : L2 => R : [H|T]) :-
    pertence(H,L1),
    !.

prove(L : [H|T] => R) :-
    !,
    prove([H|L] : T => R).

prove(L => R : [H|T]) :-
    !,
    prove(L => [H|R] : T).


transforme(n n P,P1) :-
    !,
    transforme(P,P1).

transforme((P <-> Q), ((P1 & Q1) v (n P1 & n Q1))) :-
    !,
    transforme(P,P1),
    transforme(Q,Q1).

transforme((P -> Q), (n P1 v Q1)) :-
    !,
    transforme(P,P1),
    transforme(Q,Q1).

transforme((P & Q), (P1 & Q1)) :-
    !,
    transforme(P,P1),
    transforme(Q,Q1).

transforme((P v Q), (P1 v Q1)) :-
    !,
    transforme(P,P1),
    transforme(Q,Q1).
```

```
transforme((n P), (n P1)):-
    !,
    transforme(P,P1).

transforme(P,P).


% casos onde n (negacao) aparece
regra(L : [n H|T] => R : R2,
     L : T => R : [H|R2],regra_1).

regra(L1 : L2 => R : [n H|T],
     L1 : [H|L2] => R : T ,regra_1).

% caso do & do lado esquerdo
regra(L : [H & I|T] => R,
     L : [H,I|T] => R,regra_2).

% caso do v do lado direito
regra(L => R : [H v I | T],
     L => R : [H,I|T],regra_2).


pertence(H,[H|_]) :- !.

pertence(I,[_|T]) :-
    pertence(I,T).
```

Analogamente à seção 6.1.2, serão acrescentados ao programa anterior os mesmos predicados de leitura e escrita, de maneira que a sua execução possa ser exibida. Os predicados que sofreram alguma modificação devido a tal acréscimo são listados a seguir.

### 6.3.2 Exemplos de Execução

```
mcm_mcn:-
    info_intro('MCM_MCN'),
    repeat,
```

```
    entrada(Formula),
    (Formula = fim ;
     teorema(Formula,Valor), info_valor(Valor), fail).


teorema( L => R,v) :-
     transforme(L,L1),
     transforme(R,R1),
     info_transf(L1 => R1),
     prove([]:[L1] => []:[R1]),
     !.

teorema( L => R,f) :- !.

teorema(T,v) :-
    transforme(T,T1),
    info_transf(T1),
    prove([]:[]=>[]:[T1]),
    !.

teorema(T,f).


prove(E1) :-
    regra(E1,E2,Reg),
    !,
    info_regra(E2,Reg),
    prove(E2).

% caso de v no lado esquerdo
prove(L : [H v I|T] => R) :- !,
    Esq = L : [H|T] => R,
    Dir = L : [I|T] => R,
    info_regra(Esq,Dir,regra_3),
    info_provando(Esq),
    prove(Esq),
    info_provando(Dir),
    prove(Dir).

% caso de & no lado direito
prove(L => R : [H & I|T]) :- !,
```

```
    Esq = L => R : [H|T],
    Dir = L => R : [I|T],
    info_regra(Esq,Dir,regra_4),
    info_provando(Esq),
    prove(Esq),
    info_provando(Dir),
    prove(Dir).

% caso de atomo
prove(L : [H|T] => R1 : R2) :-
    pertence(H,R1),
    !,
    info_tauto(L : [H|T] => R1 : R2).

prove(L1 : L2 => R : [H|T]) :-
    pertence(H,L1),
    !,
    info_tauto(L1 : L2 => R : [H|T]).

prove(L : [H|T] => R) :-
    !,
    prove([H|L] : T => R).

prove(L => R : [H|T]) :-
    !,
    prove(L => [H|R] : T).

prove(_) :-
    info_falha,
    fail.

% predicado adicional de escrita
info_transf(X) :-
    nl,
    write('provando a formula equivalente'),
    nl,tab(10),
    write(X),nl.
```

A seguir, é mostrada a execução de `mcm_mcn/0` usando os mesmos dados de entrada anteriores.

```
?- mcm_mcn.

    Provador de Teoremas do Calculo Proposicional
    Algoritmo de Wang - Implementacao de MCM_MCN


    Entre com a formula (fim para terminar):

p & q -> r v q & p => p -> q -> r v q & p.

provando a formula equivalente
         n (p & q) v r v q & p => n p v n q v r v q & p

regra_2   [] : [n (p & q) v r v q & p] => [] : [n p,n q v r v q & p]
regra_1   [] : [p,n (p & q) v r v q & p] => [] : [n q v r v q & p]
regra_2   [] : [p,n (p & q) v r v q & p] => [] : [n q,r v q & p]
regra_1   [] : [q,p,n (p & q) v r v q & p] => [] : [r v q & p]
regra_2   [] : [q,p,n (p & q) v r v q & p] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [p,q] : [n (p & q)] => [] : [r,q & p]
                                  e
                [p,q] : [r v q & p] => [] : [r,q & p]

Provando  [p,q] : [n (p & q)] => [] : [r,q & p]
regra_1   [p,q] : [] => [] : [p & q,r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [p,q] : [] => [] : [p,r,q & p]
                                  e
                [p,q] : [] => [] : [q,r,q & p]

Provando  [p,q] : [] => [] : [p,r,q & p]
          [p,q] : [] => [] : [p,r,q & p]    Ramo provado

Provando  [p,q] : [] => [] : [q,r,q & p]
          [p,q] : [] => [] : [q,r,q & p]    Ramo provado

Provando  [p,q] : [r v q & p] => [] : [r,q & p]
```

```
Para provar formula anterior, pela regra regra_3 devemos provar:

                [p,q] : [r] => [] : [r,q & p]
                                e
                [p,q] : [q & p] => [] : [r,q & p]

Provando  [p,q] : [r] => [] : [r,q & p]
          [r,p,q] : [] => [] : [r,q & p]     Ramo provado

Provando  [p,q] : [q & p] => [] : [r,q & p]
regra_2   [p,q] : [q,p] => [] : [r,q & p]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [p,q,p,q] : [] => [r] : [q]
                                e
                [p,q,p,q] : [] => [r] : [p]

Provando  [p,q,p,q] : [] => [r] : [q]
          [p,q,p,q] : [] => [r] : [q]     Ramo provado

Provando  [p,q,p,q] : [] => [r] : [p]
          [p,q,p,q] : [] => [r] : [p]     Ramo provado


    ** A formula e um teorema **


Entre com a formula (fim para terminar):

p v q -> p v r => p v (p -> r).

provando a formula equivalente
          n (p v q) v p v r => p v n p v r

regra_2   [] : [n (p v q) v p v r] => [] : [p,n p v r]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [] : [n (p v q)] => [] : [p,n p v r]
```

```
                              e
                  [] : [p v r] => [] : [p,n p v r]

Provando   [] : [n (p v q)] => [] : [p,n p v r]
regra_1    [] : [] => [] : [p v q,p,n p v r]
regra_2    [] : [] => [] : [p,q,p,n p v r]
regra_2    [] : [] => [p,q,p] : [n p,r]
regra_1    [] : [p] => [p,q,p] : [r]
           [] : [p] => [p,q,p] : [r]     Ramo provado

Provando   [] : [p v r] => [] : [p,n p v r]

Para provar formula anterior, pela regra regra_3 devemos provar:

                  [] : [p] => [] : [p,n p v r]
                                  e
                  [] : [r] => [] : [p,n p v r]

Provando   [] : [p] => [] : [p,n p v r]
           [p] : [] => [] : [p,n p v r]     Ramo provado

Provando   [] : [r] => [] : [p,n p v r]
regra_2    [r] : [] => [p] : [n p,r]
regra_1    [r] : [p] => [p] : [r]
           [r] : [p] => [p] : [r]     Ramo provado


    ** A formula e um teorema **


Entre com a formula (fim para terminar):

n p <-> n q <-> n r => p & r.

provando a formula equivalente
          n p & (n q & n r v n n q & n n r) v n n p &
                              n (n q & n r v n n q & n n r) => p & r


Para provar formula anterior, pela regra regra_3 devemos provar:
```

```
                [] : [n p & (n q & n r v n n q & n n r)] =>
                                                    [] : [p & r]
                                    e
                [] : [n n p & n (n q & n r v n n q & n n r)] =>
                                                    [] : [p & r]

Provando  [] : [n p & (n q & n r v n n q & n n r)] => [] : [p & r]
regra_2   [] : [n p,n q & n r v n n q & n n r] => [] : [p & r]
regra_1   [] : [n q & n r v n n q & n n r] => [] : [p,p & r]

Para provar formula anterior, pela regra regra_3 devemos provar:

                [] : [n q & n r] => [] : [p,p & r]
                                    e
                [] : [n n q & n n r] => [] : [p,p & r]

Provando  [] : [n q & n r] => [] : [p,p & r]
regra_2   [] : [n q,n r] => [] : [p,p & r]
regra_1   [] : [n r] => [] : [q,p,p & r]
regra_1   [] : [] => [] : [r,q,p,p & r]

Para provar formula anterior, pela regra regra_4 devemos provar:

                [] : [] => [p,q,r] : [p]
                                    e
                [] : [] => [p,q,r] : [r]

Provando  [] : [] => [p,q,r] : [p]    Ramo nao pode ser provado

    ** A formula nao e um teorema **


Entre com a formula (fim para terminar):

fim.
```

# 7 Medições de Tempos de Execução das Implementações e Análise dos Resultados

Preparar uma lista de problemas para testar a eficiência de Provadores Automáticos de Teoremas é uma tarefa difícil, uma vez que o que pode ser fácil para um dado Provador pode se mostrar difícil para um outro, devido à diversidade de métodos utilizados pelos Provadores.

Para a medição do tempo de execução de cada uma das implementações usamos um conjuntos de testes com 50 teoremas e 21 não teoremas, sendo que dos 50 teoremas, vários foram extraídos de [Pelletier 86] e os restantes de livros de Lógica.

Os 50 teoremas e 21 não teoremas são listados a seguir.

## 7.1 Teoremas e não Teoremas usados nas Medições

**Teoremas**

```
1     p & (p -> q) & (q -> r) => r
2     (p v q) & (p -> r) & (q ->) & (q -> r) => r
3     n q & (p -> q) => n p
4     n p & (p v r) => q
5     (p v q) & (p -> r) & (q -> r) =>
                    (p v q) & (p -> r) & (q -> r)
6     r & (n p v s) -> t v n s <->
        (r & n p -> t v n s) & (r & s -> t v n s)
7     r & t -> n q v n r & s <->
        (r & t -> n q v n r) & (r & t -> n q v s)
8     p & (r v s) -> t <->
                     (p & r -> t) & (p & s -> t)
9     p -> r & s v t <-> (p -> r v t) &
                                    (p -> s v t)
10    p & n q -> r <-> p -> r v q
11    p -> q v n r <-> p & r -> q
12    p <-> p
13    p v p <-> p v p
14    p => p v p
15    p & (p -> q) => q
16    (p -> q) & n q => n p
```

```
17    (p -> q) & (q -> r) => p -> r
18    (p v q) & n p => q
19    (p -> q) & (r -> s) & (p v r) => q v s
20    (p -> q) & (r -> s) & (n q v n s) =>
                                        n p v n r
21    p & q => p
22    p => p v q
23    p <-> n n p
24    p -> q -> r => p & q -> r
25    p & q -> r => p -> q -> r
26    (p v q) & (p -> r) & (q -> r) => r
27    p & (p -> q) & (q -> r) => r
28    p v q & n q <-> p
29    p & (q v n q) <-> p
30    p -> q <-> n p v q
31    n (p -> q) <-> p & n q
32    (p <-> q) <-> (p -> q) & (q -> p)
33    (p <-> q) <-> p & q v n p & n q
34    p -> q <-> n q -> n p
35    n n p <-> p
36    n (p -> q) => q -> p
37    n p -> q <-> n q -> p
38    p v q -> p v r => p v (q -> r)
39    p v n p
40    p v n n n p
41    (p -> q) -> p => p
42    (p v q) & (n p v q) & (p v n q) =>
                                    n (n p v n q)
43    (q -> r) & (r -> p & q) & (p -> q v r) =>
                                        p <-> q
44    p <-> p
45    ((p <-> q) <-> r) <-> p <-> q <-> r
46    (p v q & r <-> (p v q) & (p v r)
47    (p <-> q) <-> (q v n p) & (n q v p)
48    (p -> q) <-> (n p v q))
49    (p -> q) v (q -> p)
50    p & (q -> r) -> s <-> (n p v q v s) &
                                (n p v n r v s)
```

```
51    p & q -> p v r <-> p v p)
52    p & q -> p v r <-> n m)
53    p -> p & q
54    n p & (n p v q) => q)
55    p -> r & s v n t <-> (p -> n r v t) &
                                  (p <-> s & t)
56    (p v q ) & (p -> n r) & (q <-> r) =>
                (p v n r) & (p -> q) & (p -> r)
57    (p v q) & (p -> n r) & (p -> r) =>
              (p & n q) & (p -> r) & (n q -> n r)
58    ((p <-> n q) <-> n r) <-> p <-> n r <-> q)
59    p <-> n p
60    ((p <-> q) <-> p & q & (n p v n q))
61    (p <-> q) <-> (q v n p) v (n q v p)
62    (p -> q) & (r -> s) & (n s & n q) =>
                                  n p & (r & s)
63    p => p & q
64    (n p v q) & (r v n s) v (p & n n r) =>
                            ((p <-> n n q) & s)
65    n n q & n p v (n p & n n q) <-> p & q
66    q v (n n n p & n r <-> n q) <-> n q =>
                                    p & n q
67    p & n s <-> (p -> (q v s) <-> n q)
68    n q <-> (n n p -> (r v n q))
69    p v (q <-> n r) => (n p v n r)
70    n q & (r <-> n q) <-> p -> r v q
71    n p <-> n q <-> n r => p & r
```

## 7.2 Tempos de Execução dos Teoremas

As medidas de tempos de execução para as tres implementações foram realizadas em um micro computador PC-AT com co-processador aritmético. As tres implementações do algoritmo foram feitas em Arity Prolog [Arity 88] e as medidas, em centésimos de segundo, referem-se ao tempo de execução das versões interpretadas.

Deve ser ressaltado que quando se tratar de versões compiladas, o tempo de execução é, aproximadamente, 10 vezes menor.

Para os 50 teoremas foram obtidos os seguintes tempos de execução:

| Teorema | Pereira | Bratko | MCM-MCN |
|---|---|---|---|
| 1 | 20 | 35 | 12 |
| 2 | 40 | 53 | 24 |
| 3 | 9 | 21 | 7 |
| 4 | 9 | 10 | 7 |
| 5 | 205 | 1 | 76 |
| 6 | 294 | 927 | 135 |
| 7 | 232 | 788 | 230 |
| 8 | 132 | 394 | 76 |
| 9 | 146 | 451 | 79 |
| 10 | 37 | 162 | 29 |
| 11 | 37 | 161 | 31 |
| 12 | 9 | 29 | 8 |
| 13 | 20 | 29 | 16 |
| 14 | 4 | 4 | 3 |
| 15 | 9 | 15 | 6 |
| 16 | 10 | 21 | 7 |
| 17 | 21 | 65 | 13 |
| 18 | 10 | 9 | 7 |
| 19 | 50 | 67 | 26 |
| 20 | 42 | 100 | 23 |
| 21 | 5 | 4 | 3 |
| 22 | 5 | 4 | 3 |
| 23 | 12 | 47 | 9 |
| 24 | 16 | 50 | 10 |
| 25 | 16 | 63 | 10 |
| 26 | 39 | 53 | 23 |
| 27 | 20 | 35 | 12 |
| 28 | 20 | 44 | 22 |
| 29 | 21 | 61 | 18 |
| 30 | 22 | 79 | 19 |

| Teorema | Pereira | Bratko | MCM-MCN |
|---|---|---|---|
| 31 | 26 | 118 | 31 |
| 32 | 98 | 257 | 86 |
| 33 | 101 | 194 | 94 |
| 34 | 26 | 116 | 22 |
| 35 | 12 | 46 | 9 |
| 36 | 8 | 21 | 6 |
| 37 | 26 | 111 | 21 |
| 38 | 23 | 43 | 13 |
| 39 | 4 | 7 | 3 |
| 40 | 5 | 13 | 3 |
| 41 | 9 | 17 | 7 |
| 42 | 79 | 6 5 | 38 |
| 43 | 146 | 183 | 52 |
| 44 | 9 | 29 | 9 |
| 45 | 1303 | 894 | 615 |
| 46 | 57 | 123 | 42 |
| 47 | 97 | 212 | 89 |
| 48 | 22 | 79 | 18 |
| 49 | 8 | 29 | 6 |
| 50 | 141 | 380 | 84 |

De uma maneira simplificada, a implementação de Pereira deve chegar, para provar o teorema, a nível de átomos da fórmula, enquanto que a de Bratko faz a prova baseada em busca de padrões.

Ainda que a busca de padrões seja uma operação com alto consumo de tempo, é evidente que se esta busca de padrões tiver sucesso logo no início da prova, a implementação de Bratko tende a ser mais eficiente. Isto é evidenciado pelos tempos de execução obtidos.

Pode ser observado que a implementação de Bratko é mais eficiente que a de Pereira nos casos dos teoremas de números 5, 18, 21, 22, 42 e 45.

O caso do teorema de número 5 é atípico, uma vez que neste teorema, exatamente a mesma fórmula comparece em ambos os lados do símbolo => e ela é imediatamente detectada na implementação de Bratko.

Nos outros cinco casos em que a implementação de Bratko supera a de Pereira, a detecção do padrão é quase imediata, após as primeiras transformações.

Considerando agora a nossa implementação, Bratko a supera somente no caso

atípico do teorema de número 5, mostrado novamente a seguir.

```
(p v q) & (p -> r) & (q -> r) -> r =>
                    (p v q) & (p -> r) & (q -> r) -> r)
```

Porém, observe que uma troca de ordem de sub-fórmulas, que não provoca qualquer alteração da fórmula lógica, tal como:

```
(p v q) & (q -> r) & (p -> r) -> r =>
                    (p v q) & (p -> r) & (q -> r) -> r)
```

provoca um mal desempenho da implementação de Bratko, pelas razões já comentadas acima. Neste caso particular, foram obtidos os seguintes tempos de execução:

| Pereira | Bratko | MCM-MCN |
|---|---|---|
| 206 | 213 | 78 |

Quanto à implementação de Pereira, ela é ligeiramente mais eficiente com relação à nossa somente no caso do teorema de número 31. Neste caso, as transformações iniciais de implicações e equivalências realizadas por nossa implementação mostraram-se desvantajosas.

A tabela a seguir mostra que, em média, a nossa implementação é mais eficiente que a de Bratko por um fator de 3.2 e mais eficiente que a de Pereira por um fator de 1.6.

| Tempos cent. seg. | Pereira | Bratko | MCM-MCN |
|---|---|---|---|
| Total | 3712 | 6719 | 2192 |
| T. Médio | 74 | 134 | 44 |

## 7.3 Tempos de Execução dos Não Teoremas

Para os 21 não teoremas foram obtidos os seguintes tempos de execução:

| Não Teorema | Pereira | Bratko | MCM-MCN |
|---|---|---|---|
| 51 | 9 | 43 | 12 |
| 52 | 9 | 40 | 11 |
| 53 | 8 | 23 | 6 |
| 54 | 6 | 15 | 6 |
| 55 | 27 | 89 | 34 |
| 56 | 101 | 86 | 41 |
| 57 | 105 | 44 | 49 |
| 58 | 31 | 182 | 88 |
| 59 | 6 | 27 | 6 |
| 60 | 11 | 45 | 43 |
| 61 | 53 | 85 | 25 |
| 62 | 14 | 97 | 13 |
| 63 | 7 | 7 | 5 |
| 64 | 18 | 62 | 13 |
| 65 | 10 | 44 | 32 |
| 66 | 431 | 90 | 17 |
| 67 | 18 | 108 | 11 |
| 68 | 18 | 89 | 9 |
| 69 | 7 | 17 | 7 |
| 70 | 52 | 136 | 18 |
| 71 | 39 | 52 | 9 |

A tabela a seguir mostra que, em média, a nossa implementação é mais eficiente que a de Bratko por um fator de 3 e mais eficiente que a de Pereira por um fator de 2.

Observe que o tempo excessivo de execução da implementação de Pereira para o não teorema de número 66 deve-se, principalmente, ao tempo gasto na simplificação de negações.

| Tempo cent. seg. | Pereira | Bratko | MCM-MCN |
|---|---|---|---|
| Total | 980 | 1381 | 455 |
| T. Médio | 45 | 66 | 22 |

# 8 Conclusões

Prova Automática de Teoremas é importante na área de Inteligência Artificial - IA - para resolver problemas nos quais o conhecimento pode ser axiomatizado. Portanto, há um grande interesse em Provadores computacionalmente eficientes.

Apresentamos neste trabalho tres implementações em Prolog — sendo a última de nossa autoria — do Algoritmo de Wang, para prova sintática de teoremas do Cálculo Proposicional.

A linguagem Prolog é bastante popular na comunidade de IA pois possui características apropriadas para o desenvolvimento de programas nesta área. Prolog está relacionada com Lógica Matemática e sua sintaxe e semântica são fortemente baseadas em Lógica.

A habilidade que Prolog possui para resolver muitos dos detalhes procedimentais é considerada uma de suas vantagens. Porém, o aspecto declarativo não é sempre suficiente para a realização de implementações onde o tempo de execução seja crítico. O conhecimento do aspecto procedimental da linguagem permite a realização de implementações eficientes, como pode ser verificado pelos resultados deste trabalho.

Observe que um Provador de Teoremas é, frequentemente, um pequeno módulo dentro de um sistema computacional de IA que será ativado inúmeras vezes. Portanto, o seu tempo de execução deve ser, sempre que possível, minimizado.

Mostramos neste trabalho que, sem comprometer o aspecto declarativo da linguagem Prolog, é possível desenvolver programas eficientes levando em conta, entre outros, os mecanismos internos da linguagem. Aplicando este conhecimento, conseguimos desenvolver uma implementação do Algoritmo de Wang que é sensivelmente mais eficiente que outras duas implementações largamente conhecidas na literatura.

Ainda nesta linha de pesquisa de métodos sintáticos de prova de teoremas, pretendemos, oportunamente, abordar outros métodos sintáticos, possivelmente resolução, de maneira a poder apontar os mais eficientes sob o ponto de vista computacional.

# Referências


[Arity 88] Arity-Prolog Programming Language, version 5.1, 1988

[Coelho 88] Coelho,H.; Cotta,J.C. *Prolog by Examples.* Springer-Verlag, 1988

[Pelletier 86] Pelletier,F.J. *Seventy-Five Problems for Testing Automatic Theorem Provers.* Journal of Automated Reasoning 2, pp 191-216, 1986

[Wang 60] Wang,H. *Towards Mechanical Mathematics.* IBM Journal of Research and Development, Vol 4, pp 2-22,1960

[Wang 64] Wang,H. *A Survey of Mathematical Logic.* North-Holland Publishing Company, 1964

[Tanimoto 87] Tanimoto, S.L. *The Elements of Artificial Intelligence.* Computer Science Press, 1987

**NOTAS DO ICMSC/USP**

Nº 61 - GIONGO, M.A.; TÁBOAS, P.Z. - Roses play a role in some inverse problems from bifurcation theory

Nº 60 - MORABITO, R.N.; ARENALES, M.N.; ARCARO, V.F. - An and-or-graph representation for two-dimensional cutting problems

Nº 59 - ACHCAR, J.A. - A bayesian approach to reparametrization of the exponential distribution with type I censored data

Nº 58 - ACHCAR, J.A. - An useful reparametrization for the extrem value distribution

Nº 57 - MASIERO, P.C.; et al - Um ambiente de desenvolvimento baseado na abordagem operacional

Nº 56 - ANDRADE, E.X.L.; BRACCIALI, C.F. - Um método, assemelhado ao de Francis, para determinação de auto-valores de matrizes

Nº 55 - BRUCE, J.W. - Euler characteristics of real varieties

Nº 54 - FURKOTTER, M.; RODRIGUES, H.M - Symmetry and bifurcation to 2"-periodic solutions on nonlinear second order equations with 2"/m-periodic forcings

Nº 53 - RIEGER, J.H.; RUAS, M.A.S. - Classification of A-simple germs from $K^n$ to $K^2$

Nº 52 - FURKOTTER, M.; RODRIGUES, H.M. - On harmonic and subharmonic solutions of nonlinear second order equations: symmetry and bifurcation

Nº 51 - MONARD, M.C.; RODRIGUES, S.R. - Implementação lógica de um motor de inferência com raciocínio "backward chaining" para a construção de sistemas especialistas