

Métodos de Prova

- Os métodos de prova agrupam-se em dois tipos:
- **Aplicação de regras de inferência**
 - Geração legítima (sólida) de novas proposições a partir de antigas
 - Prova = uma sequência de aplicação de regras de inferência
 - Regras de inferência podem ser operadores em algoritmos de procura
 - Habitualmente obrigam à tradução das frases para uma forma normal
- **Verificação de modelos**
 - enumeração por tabelas de verdade (sempre exponencial em n)
 - melhoramentos ao retrocesso, e.g Davis-Putnam-Logemann-Loveland
 - procura heurística em espaço de modelos (sólido mas incompleto)
 - e.g., algoritmos trepa-colinas com min-conflitos

Resolução

- **Forma Normal Conjuntiva** (FNC – universal)
 - **Conjunção** de **disjunções** de literais
 - disjunção de literais = **cláusula**
 - E.g., $(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$
- Regra de inferência **Resolução**:

$$\frac{l_1 \vee \dots \vee l_k, \quad m_1 \vee \dots \vee m_n}{l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n}$$

em que l_i e m_j são literais complementares. E.g.,

$$\frac{P_{1,3} \vee P_{2,2}, \quad \neg P_{2,2}}{P_{1,3}}$$

- Resolução **é sólida e completa** para a lógica proposicional
 - Apenas deriva frases verdadeiras
 - Pode sempre ser usada para confirmar ou refutar uma frase

Conversão para FNC

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

1. Eliminar $\Leftrightarrow$ substituindo $\alpha \Leftrightarrow \beta$ por $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
 $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
2. Eliminar $\Rightarrow$ substituindo $\alpha \Rightarrow \beta$ por $\neg \alpha \vee \beta$
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg (P_{1,2} \vee P_{2,1}) \vee B_{1,1})$
3. Deslocar $\neg$ para dentro recorrendo às leis de De Morgan e absorção da dupla negação:
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$
4. Aplicar distributividade ($\vee$ sobre $\wedge$):
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

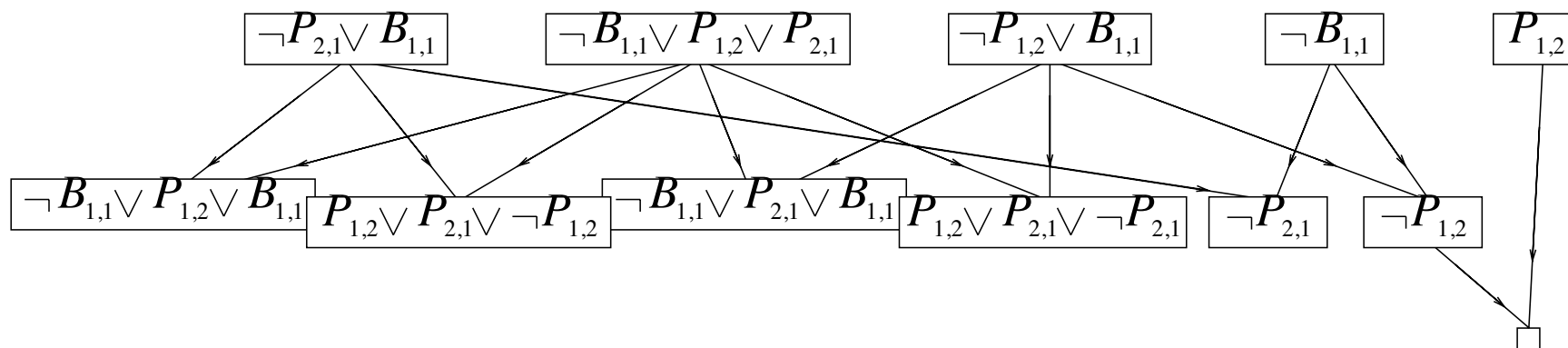
Algoritmo de Resolução

- Prova por contradição, i.e., demonstrar que $KB \wedge \neg\alpha$ é insatisfazível

```
function PL-RESOLUTION( $KB, \alpha$ ) returns true or false  
   $clauses \leftarrow$  the set of clauses in the CNF representation of  $KB \wedge \neg\alpha$   
   $new \leftarrow \{\}$   
  loop do  
    for each  $C_i, C_j$  in  $clauses$  do  
       $resolvents \leftarrow$  PL-RESOLVE( $C_i, C_j$ )  
      if  $resolvents$  contains the empty clause then return true  
       $new \leftarrow new \cup resolvents$   
  if  $new \subseteq clauses$  then return false  
   $clauses \leftarrow clauses \cup new$ 
```

Exemplo de Resolução

- $KB = B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) \wedge \neg B_{1,1} \quad \alpha = \neg P_{1,2}$
- $KB \wedge \neg \alpha$ na FNC:
 - $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \wedge \neg B_{1,1} \wedge P_{1,2}$



Encadeamento para a frente e para trás

- A completude da resolução tornam-no num modelo de inferência muito importante
- Uma parte significativa do conhecimento no mundo real apenas necessita de cláusulas de uma forma restrita:
- **Cláusulas na Forma de Horn (ou Cláusulas de Horn)**

- Cláusula de Horn = disjunção de literais com, no máximo, um literal positivo
 - $\neg\alpha_1 \vee \dots \vee \neg\alpha_n \vee \beta$
- Podem ser re-escritas como uma implicação
 - $\alpha_1 \wedge \dots \wedge \alpha_n \Rightarrow \beta$
- **Modus Ponens** é completa para KBs de Horn:

$$\frac{\alpha_1, \dots, \alpha_n, \quad \alpha_1 \wedge \dots \wedge \alpha_n \Rightarrow \beta}{\beta}$$

- Pode ser utilizada com **encadeamento para a frente** ou **para trás**.
- Estes algoritmos são muito naturais e executam em tempo linear.

Encadeamento para a frente

- **Ideia:** disparar toda a regra cujas premissas estão satisfeitas na KB,
- adicionar a sua conclusão à KB, até se chegar à pergunta

$$P \Rightarrow Q$$

$$L \wedge M \Rightarrow P$$

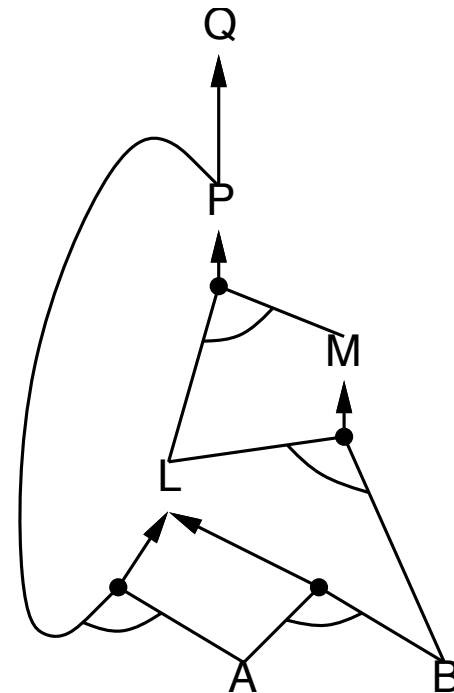
$$B \wedge L \Rightarrow M$$

$$A \wedge P \Rightarrow L$$

$$A \wedge B \Rightarrow L$$

A

B



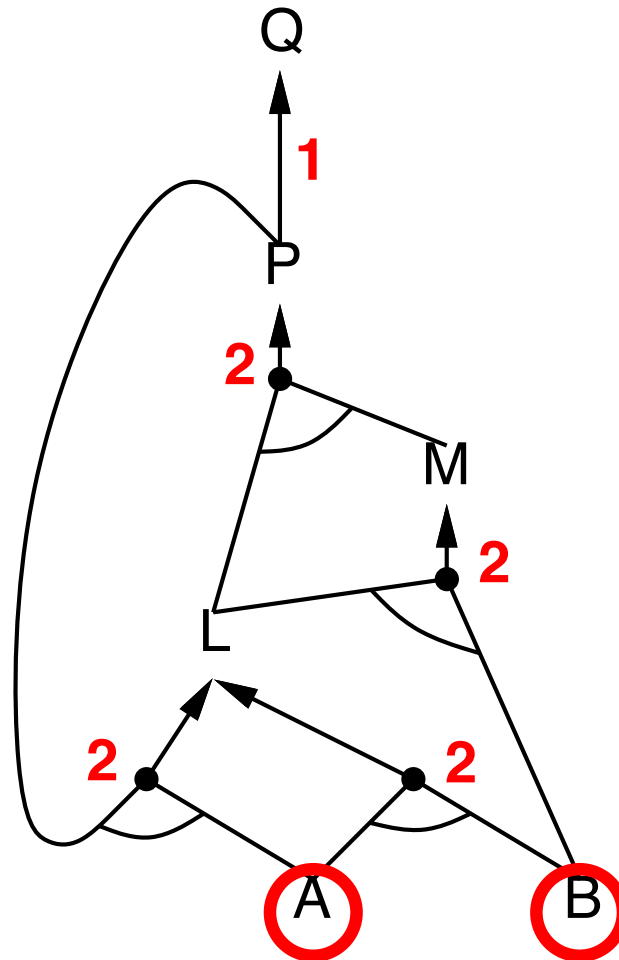
Algoritmo de encadeamento para a frente

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  local variables: count, a table, indexed by clause, initially the number of premises
                  inferred, a table, indexed by symbol, each entry initially false
                  agenda, a list of symbols, initially the symbols known to be true

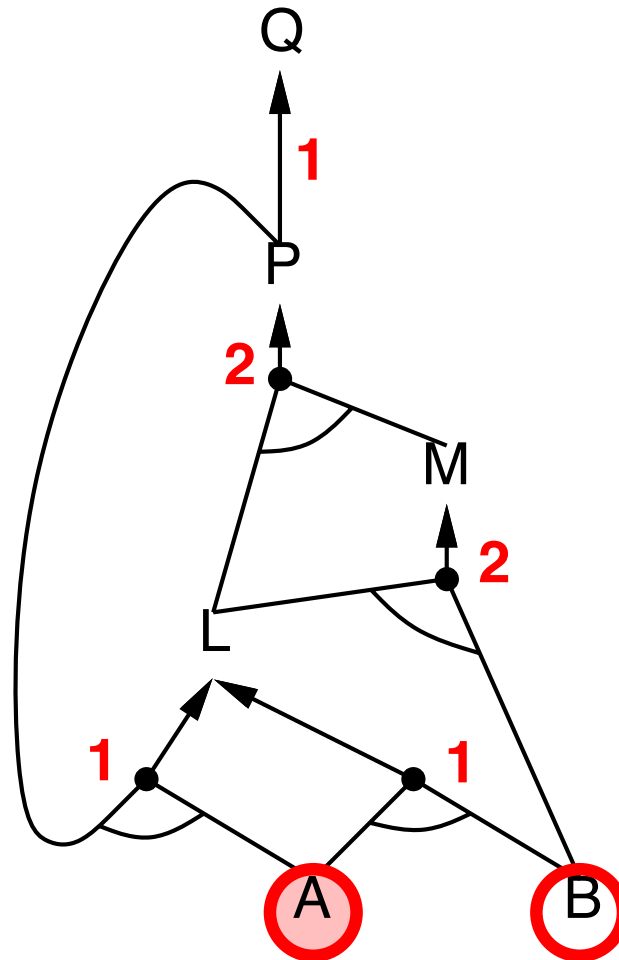
  while agenda is not empty do
    p ← POP(agenda)
    unless inferred[p] do
      inferred[p] ← true
      for each Horn clause c in whose premise p appears do
        decrement count[c]
        if count[c] = 0 then do
          if HEAD[c] = q then return true
          PUSH(HEAD[c], agenda)

  return false
```

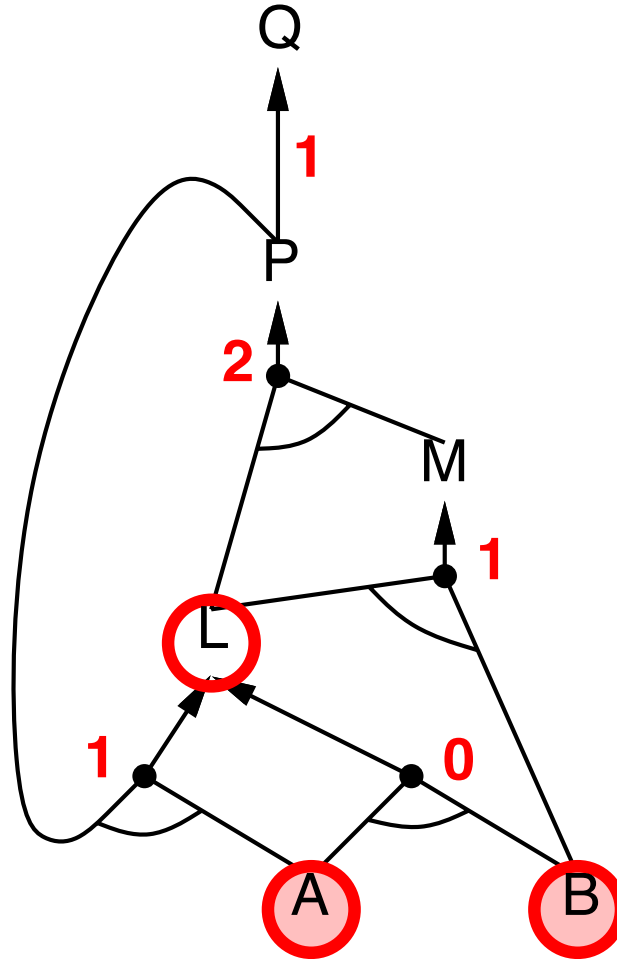
Exemplo de encadeamento para a frente



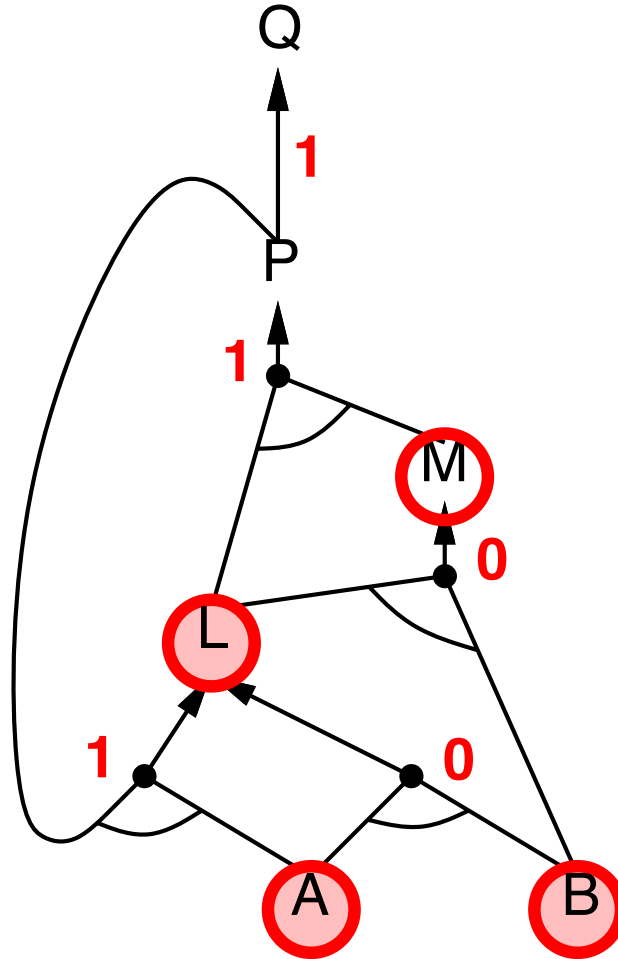
Exemplo de encadeamento para a frente



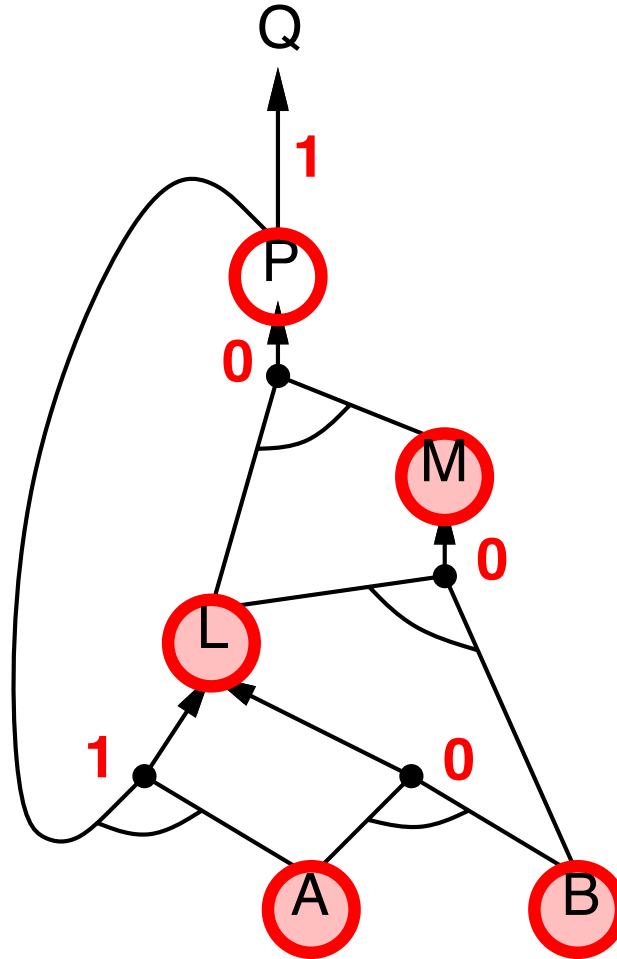
Exemplo de encadeamento para a frente



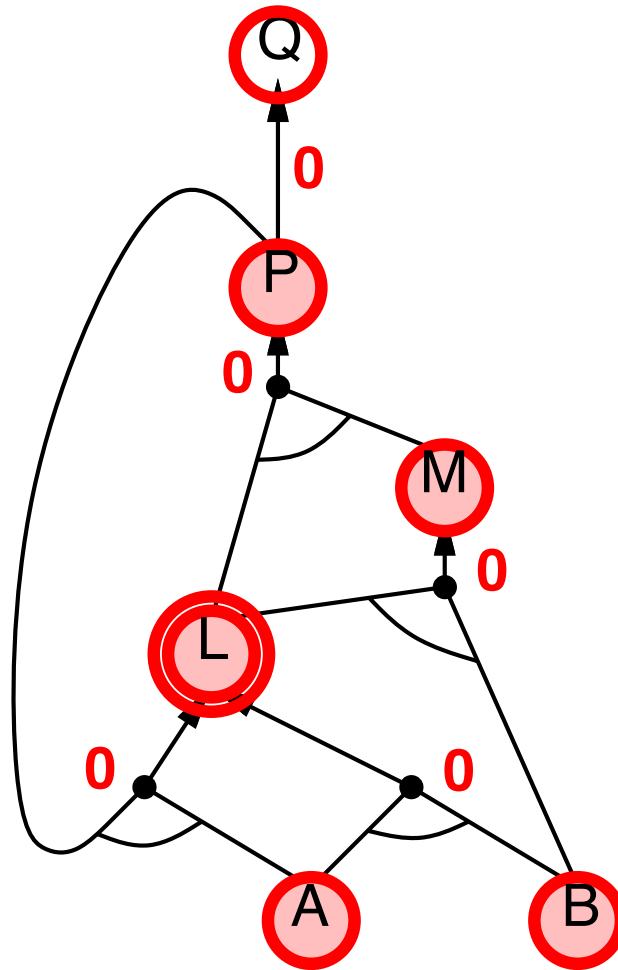
Exemplo de encadeamento para a frente



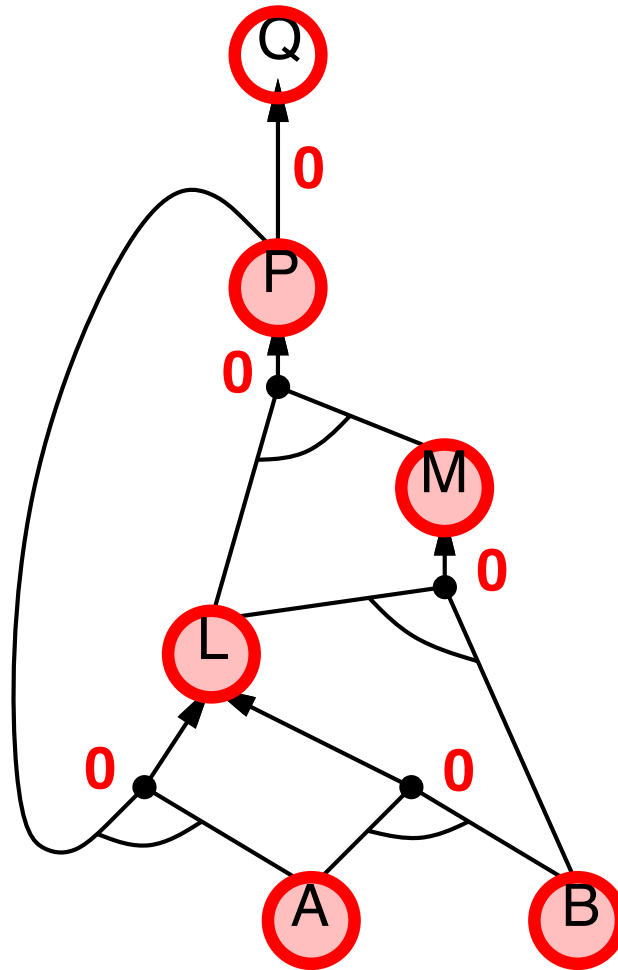
Exemplo de encadeamento para a frente



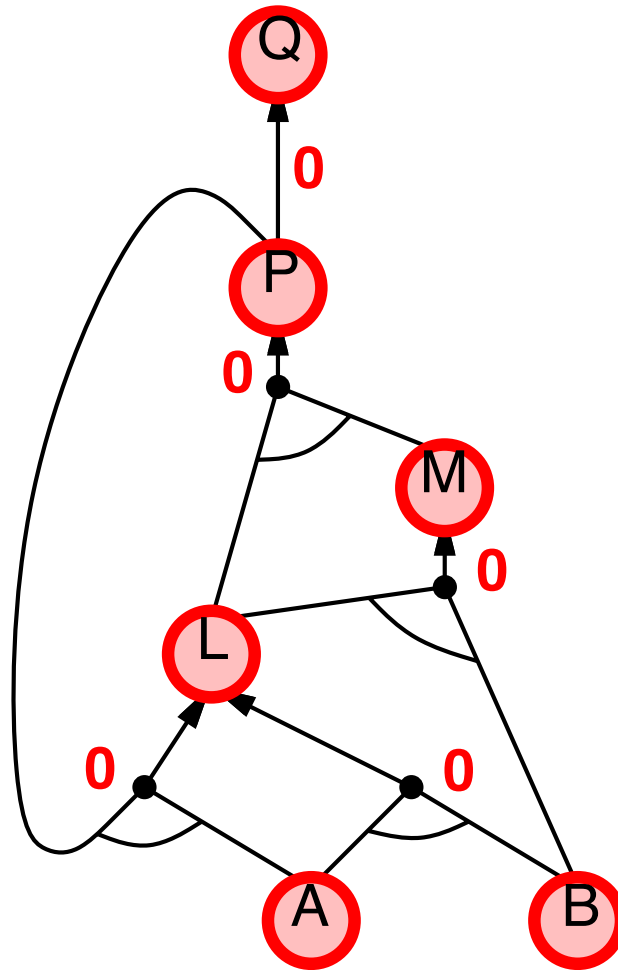
Exemplo de encadeamento para a frente



Exemplo de encadeamento para a frente



Exemplo de encadeamento para a frente



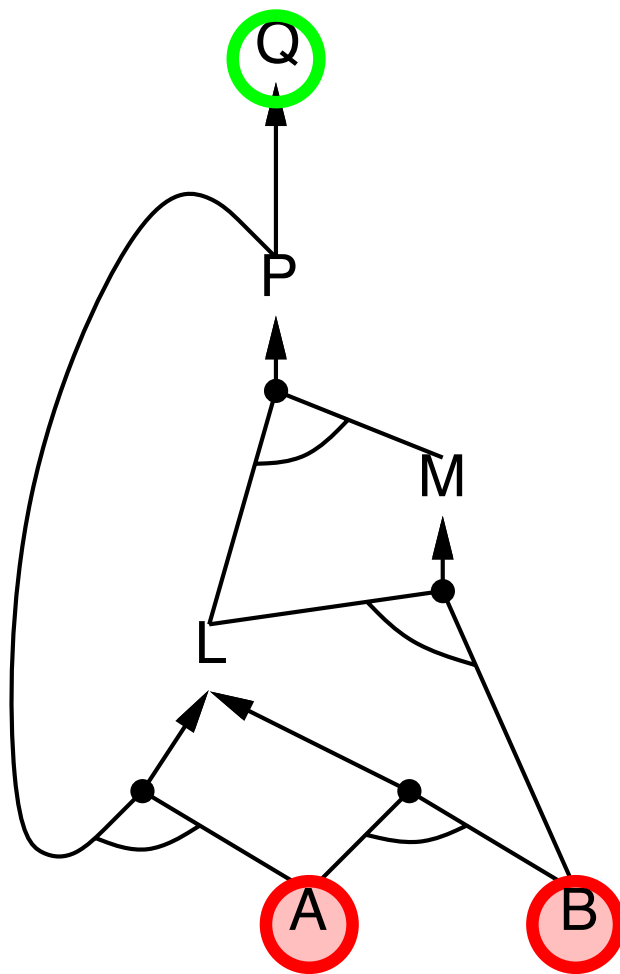
Demonstração de completude

- EF deriva toda a proposição atômica que é concluída a partir de KB
- 1. EF atinge um ponto fixo em que não são derivadas novas proposições atômicas
- 2. Considere-se o estado final como um modelo m , atribuindo verdadeiro/falso aos símbolos
- 3. Toda a cláusula em KB inicial é verdadeira em m
 - **Demo:** Suponha-se que a cláusula $\alpha_1 \wedge \dots \wedge \alpha_k \Rightarrow \beta$ é falsa em m
 - Logo $\alpha_1 \wedge \dots \wedge \alpha_k$ é verdadeiro em m e β é falso em m
 - Logo o algoritmo não atingiu um ponto fixo!
- 4. Portanto m é modelo de KB
- 5. Se $KB \models q$, q é verdadeiro em todo o modelo de KB, incluindo m

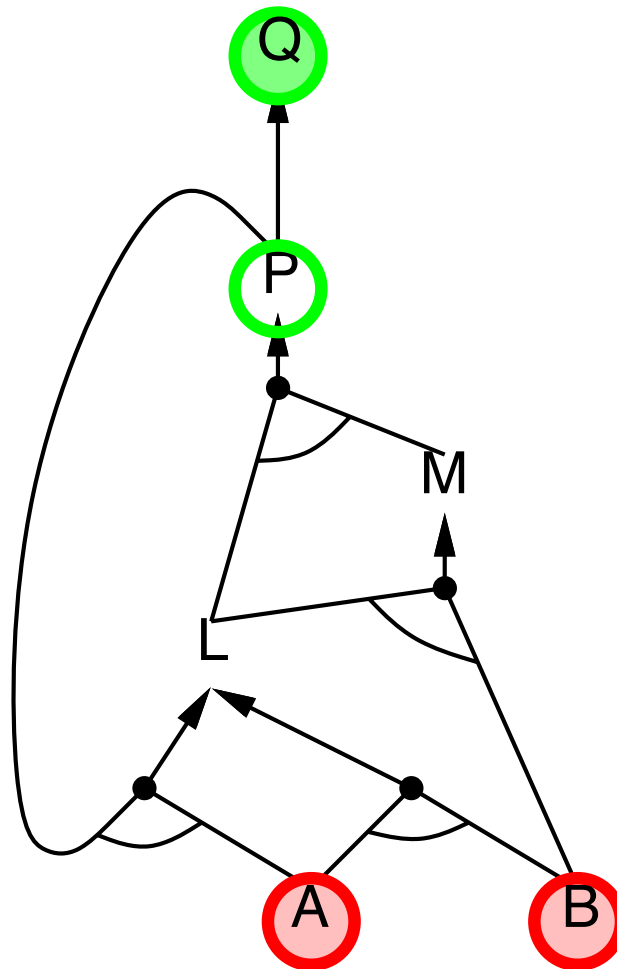
Encadeamento para trás

- **Ideia:** andar ao contrário a partir da pergunta q :
 - para provar q por ET,
 - verificar se q já é sabido, ou
 - provar por ET todas as premissas de alguma regra concluindo q
- Evitar ciclos: verificar se um novo sub-objectivo já se encontra na pilha de objectivos
- Evitar trabalho repetido: verificar se o novo sub-objectivo
 1. já foi demonstrado verdadeiro, ou
 2. já falhou

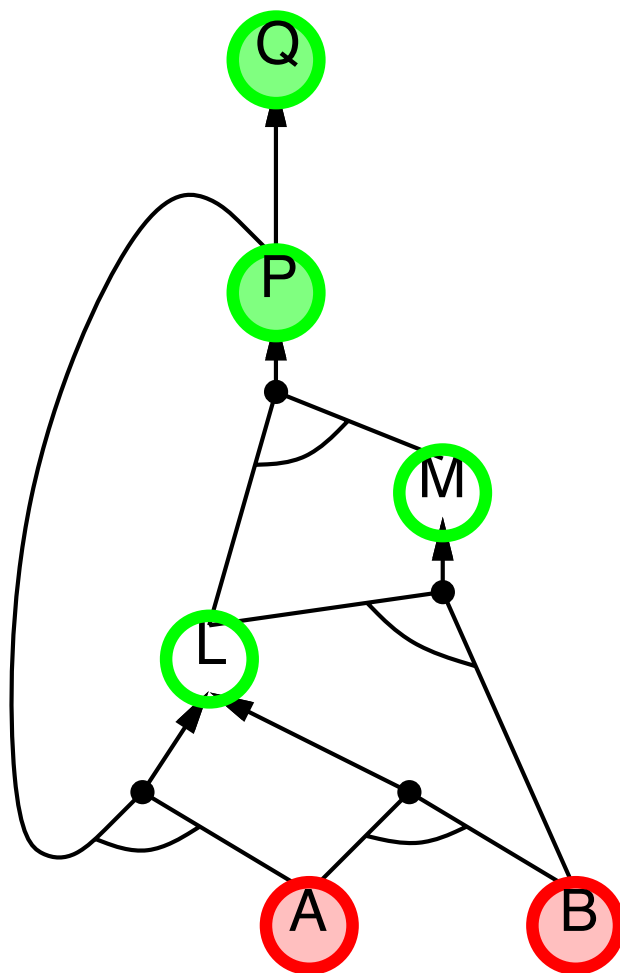
Exemplo de encadeamento para trás



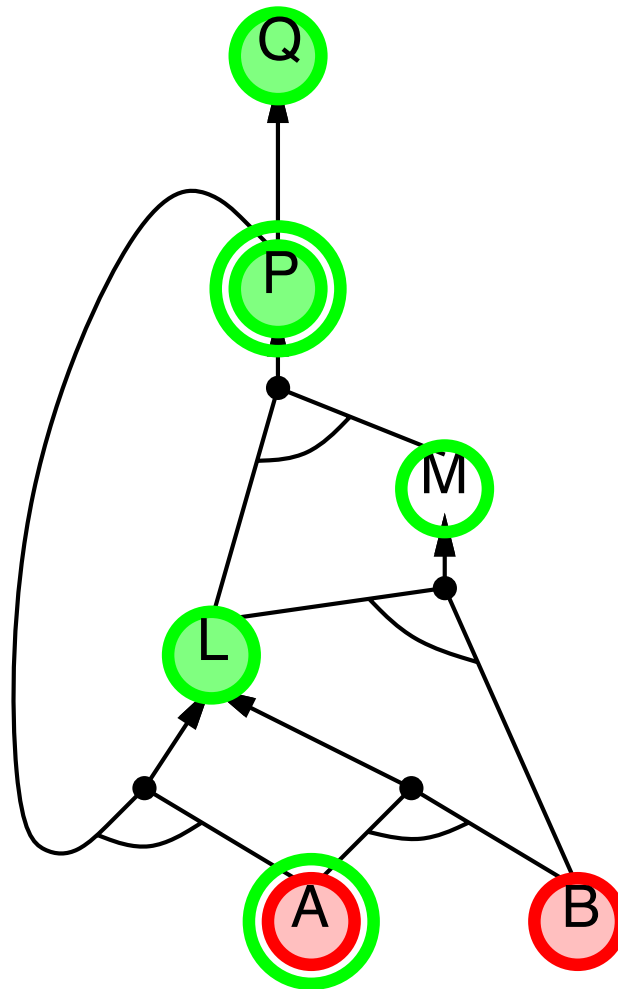
Exemplo de encadeamento para trás



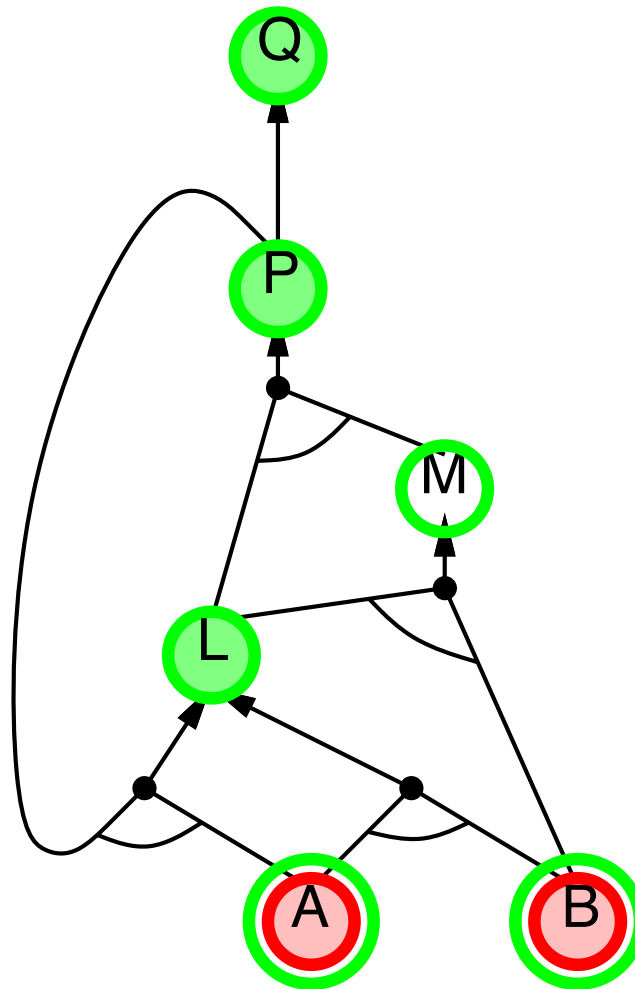
Exemplo de encadeamento para trás



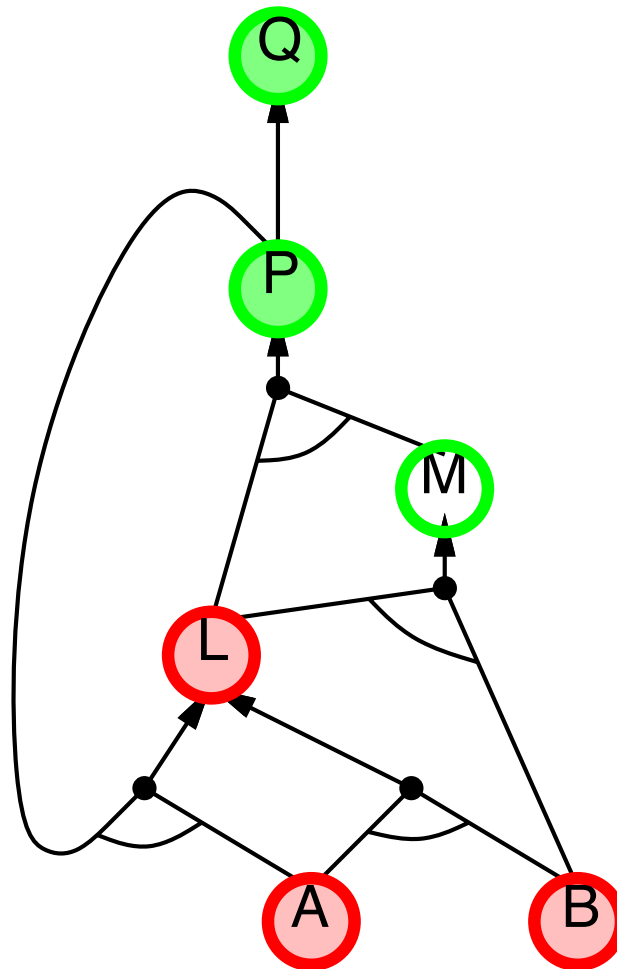
Exemplo de encadeamento para trás



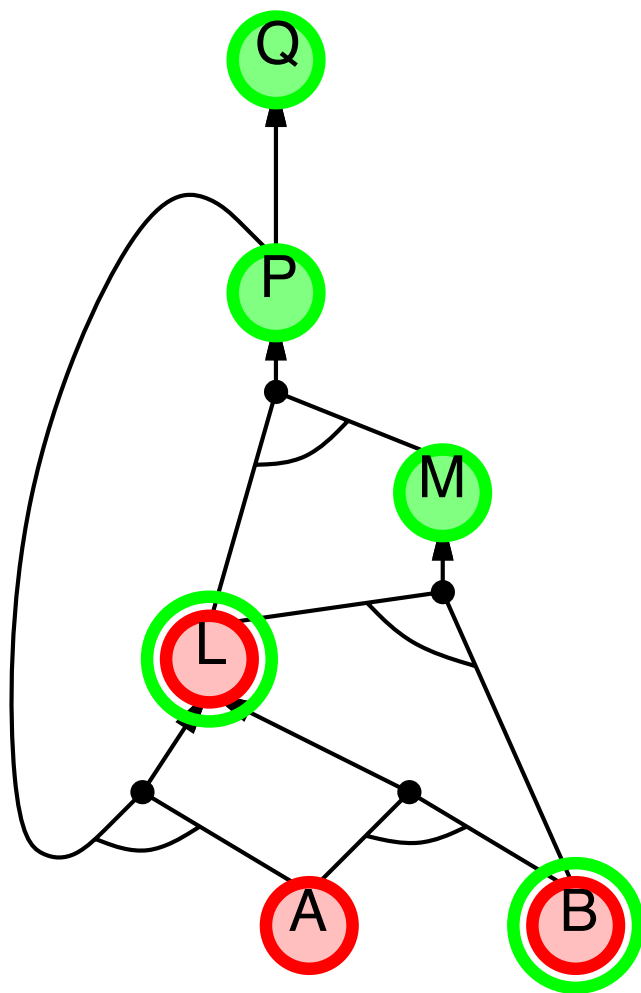
Exemplo de encadeamento para trás



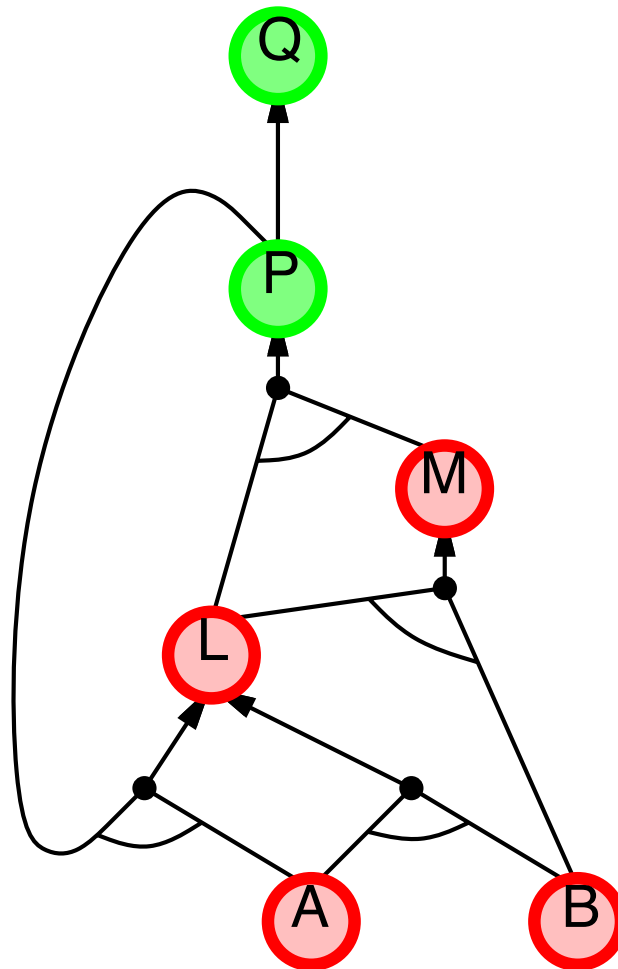
Exemplo de encadeamento para trás



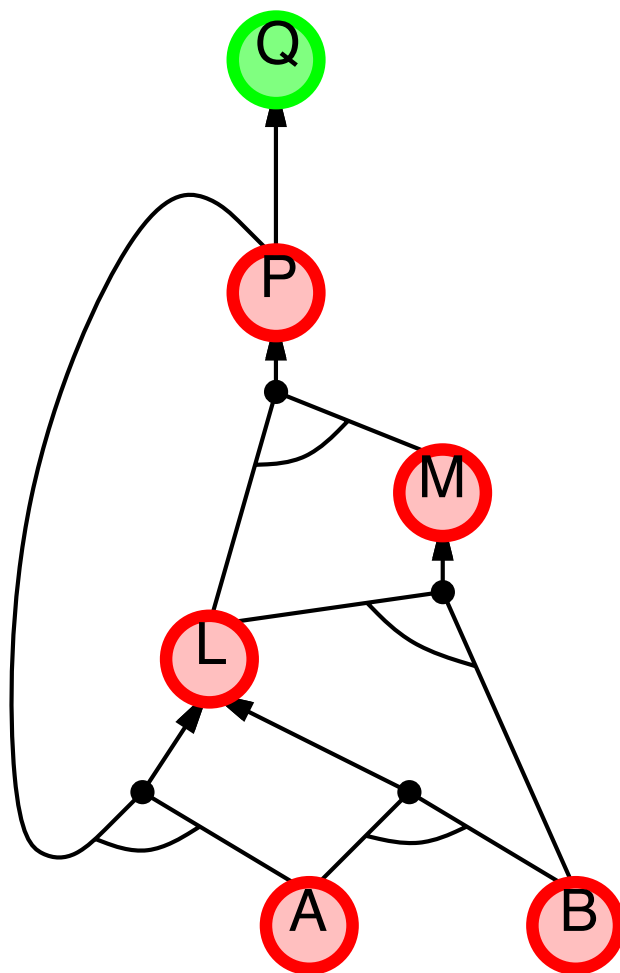
Exemplo de encadeamento para trás



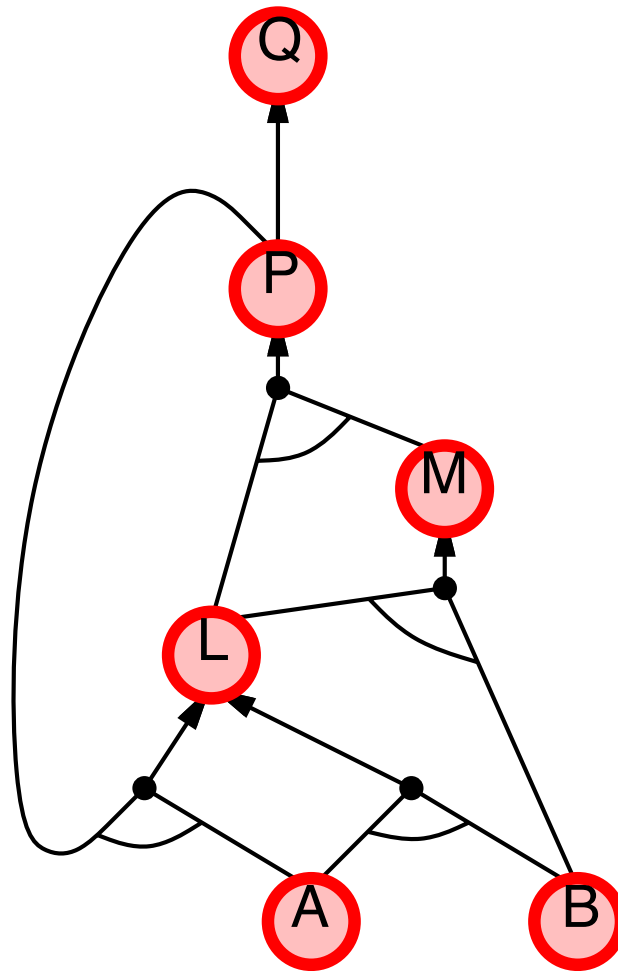
Exemplo de encadeamento para trás



Exemplo de encadeamento para trás



Exemplo de encadeamento para trás



Encadeamento para a frente vs. para trás

- EF é **guiado pelos dados**, cf. processamento automático, inconsciente, e.g., reconhecimento de objectos, decisões rotineiras
- Pode efectuar muito trabalho desnecessário que é irrelevante para o objectivo
- ET é **guiado pelo objectivo**, adequado para a resolução de problemas, e.g.,
 - Onde estão as minhas chaves?
 - Como posso fazer um doutoramento?
- Complexidade do ET pode ser **muito inferior** do que linear no tamanho da KB

Verificação eficiente de satisfatibilidade

- Existem algoritmos eficientes de satisfatibilidade, baseados em verificação de modelos, que nos permitem lidar com problemas combinatórios complexos
 - Todos os problemas NP-completos podem ser reduzidos ao problema da satisfatibilidade de cláusulas de lógica proposicional
- Duas famílias de algoritmos:
 - Baseados em retrocesso, e.g. Davis-Putnam-Logemann-Loveland
 - Por melhoramento iterativo (procura local), e.g. WalkSAT

Algoritmo DPLL

- Enumeração recursiva em profundidade primeiro de todos os modelos possíveis para proposições na FNC, com as seguintes melhorias em relação à enumeração por tabelas de verdade:
 - **Terminação com modelos parciais**: uma cláusula é verdadeira quando pelo menos um dos literais é verdadeiro. Logo, os restantes valores dos símbolos proposicionais são irrelevantes.
 - **Heurística dos símbolos puros**: um símbolo é puro quando ocorre sempre com o mesmo sinal em todas as cláusulas. Nas proposições abaixo, A e $\neg B$ são puros
$$(A \vee \neg B) \wedge (\neg B \vee \neg C) \wedge (C \vee A)$$
 - **Heurística da cláusula unitária**: Quando todos os literais são falsos à exceção de um, o valor desse fica automaticamente definido. Pode originar propagações unitárias em cascata.

Algoritmo de Davis-Putnam

function DPLL-SATISFIABLE?(*s*) **returns** *true* or *false*

inputs: *s*, a sentence in propositional logic

clauses $\leftarrow$ the set of clauses in the CNF representation of *s*

symbols $\leftarrow$ a list of the proposition symbols in *s*

return DPLL(*clauses*, *symbols*, [])

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

if every clause in *clauses* is true in *model* **then return** *true*

if some clause in *clauses* is false in *model* **then return** *false*

P, *value* $\leftarrow$ FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols* - *P*, [*P* = *value* | *model*])

P, *value* $\leftarrow$ FIND-UNIT-CLAUSE(*clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols* - *P*, [*P* = *value* | *model*])

P $\leftarrow$ FIRST(*symbols*); *rest* $\leftarrow$ REST(*symbols*)

return DPLL(*clauses*, *rest*, [*P* = *true* | *model*]) **or** DPLL(*clauses*, *rest*, [*P* = *false* | *model*])

Algoritmo de Davis-Putnam

Verifica se todas as cláusulas são verdadeiras. Se forem, s é satisfazível.

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*

inputs: s , a sentence in propositional logic

$clauses \leftarrow$ the set of clauses in the CNF representation of s

$symbols \leftarrow$ a list of the proposition symbols in s

return DPLL($clauses, symbols, []$)

function DPLL($clauses, symbols, model$) **returns** *true* or *false*

if every clause in $clauses$ is true in $model$ **then return** *true*

if some clause in $clauses$ is false in $model$ **then return** *false*

$P, value \leftarrow$ FIND-PURE-SYMBOL($symbols, clauses, model$)

if P is non-null **then return** DPLL($clauses, symbols-P, [P = value|model]$)

$P, value \leftarrow$ FIND-UNIT-CLAUSE($clauses, model$)

if P is non-null **then return** DPLL($clauses, symbols-P, [P = value|model]$)

$P \leftarrow$ FIRST($symbols$); $rest \leftarrow$ REST($symbols$)

return DPLL($clauses, rest, [P = true|model]$) **or** DPLL($clauses, rest, [P = false|model]$)

Algoritmo de Davis-Putnam

Verifica se alguma cláusula é falsa. Se for, s é insatisfazível.

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*

inputs: s , a sentence in propositional logic

$clauses \leftarrow$ the set of clauses in the CNF representation of s

$symbols \leftarrow$ a list of the proposition symbols in s

return DPLL($clauses, symbols, []$)

function DPLL($clauses, symbols, model$) **returns** *true* or *false*

if every clause in $clauses$ is true in $model$ **then return** *true*

if some clause in $clauses$ is false in $model$ **then return** *false*

$P, value \leftarrow$ FIND-PURE-SYMBOL($symbols, clauses, model$)

if P is non-null **then return** DPLL($clauses, symbols - P, [P = value | model]$)

$P, value \leftarrow$ FIND-UNIT-CLAUSE($clauses, model$)

if P is non-null **then return** DPLL($clauses, symbols - P, [P = value | model]$)

$P \leftarrow$ FIRST($symbols$); $rest \leftarrow$ REST($symbols$)

return DPLL($clauses, rest, [P = true | model]$) **or** DPLL($clauses, rest, [P = false | model]$)

Algoritmo de Davis-Putnam

Para um literal puro,
atribui o valor de
verdade
correspondente:
T se literal positivo
F se literal negativo

function DPLL-SATISFIABLE?(*s*) **returns** *true* or *false*

inputs: *s*, a sentence in propositional logic

clauses $\leftarrow$ the set of clauses in the CNF representation of *s*

symbols $\leftarrow$ a list of the proposition symbols in *s*

return DPLL(*clauses*, *symbols*, [])

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

if every clause in *clauses* is true in *model* **then return** *true*

if some clause in *clauses* is false in *model* **then return** *false*

P, *value* $\leftarrow$ FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols*−*P*, [*P* = *value* | *model*])

P, *value* $\leftarrow$ FIND-UNIT-CLAUSE(*clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols*−*P*, [*P* = *value* | *model*])

P $\leftarrow$ FIRST(*symbols*); *rest* $\leftarrow$ REST(*symbols*)

return DPLL(*clauses*, *rest*, [*P* = *true* | *model*]) **or** DPLL(*clauses*, *rest*, [*P* = *false* | *model*])

Algoritmo de Davis-Putnam

Para uma cláusula unitária, atribui o valor de verdade ao literal:

T se literal positivo
F se literal negativo

function DPLL-SATISFIABLE?(*s*) **returns** *true* or *false*

inputs: *s*, a sentence in propositional logic

clauses ← the set of clauses in the CNF representation of *s*

symbols ← a list of the proposition symbols in *s*

return DPLL(*clauses*, *symbols*, [])

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

if every clause in *clauses* is true in *model* **then return** *true*

if some clause in *clauses* is false in *model* **then return** *false*

P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols*−*P*, [*P* = *value* | *model*])

P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols*−*P*, [*P* = *value* | *model*])

P ← FIRST(*symbols*); *rest* ← REST(*symbols*)

return DPLL(*clauses*, *rest*, [*P* = *true* | *model*]) **or** DPLL(*clauses*, *rest*, [*P* = *false* | *model*])

Algoritmo de Davis-Putnam

Escolhe uma variável sem valor de verdade atribuído e, recursivamente, chama o DPLL para ambos os casos i.e. atribuindo o valor T ou F.

function DPLL-SATISFIABLE?(*s*) **returns** *true* or *false*

inputs: *s*, a sentence in propositional logic

clauses $\leftarrow$ the set of clauses in the CNF representation of *s*

symbols $\leftarrow$ a list of the proposition symbols in *s*

return DPLL(*clauses*, *symbols*, [])

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

if every clause in *clauses* is true in *model* **then return** *true*

if some clause in *clauses* is false in *model* **then return** *false*

P, *value* $\leftarrow$ FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols* - *P*, [*P* = *value* | *model*])

P, *value* $\leftarrow$ FIND-UNIT-CLAUSE(*clauses*, *model*)

if *P* is non-null **then return** DPLL(*clauses*, *symbols* - *P*, [*P* = *value* | *model*])

P $\leftarrow$ FIRST(*symbols*); *rest* $\leftarrow$ REST(*symbols*)

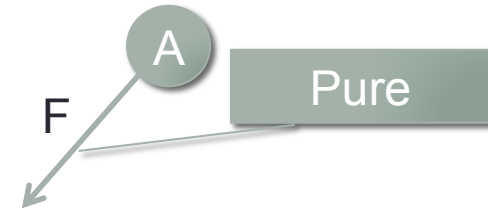
return DPLL(*clauses*, *rest*, [*P* = *true* | *model*]) **or** DPLL(*clauses*, *rest*, [*P* = *false* | *model*])

Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$

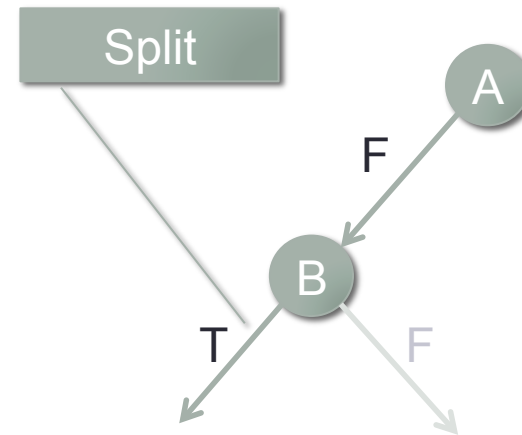
Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



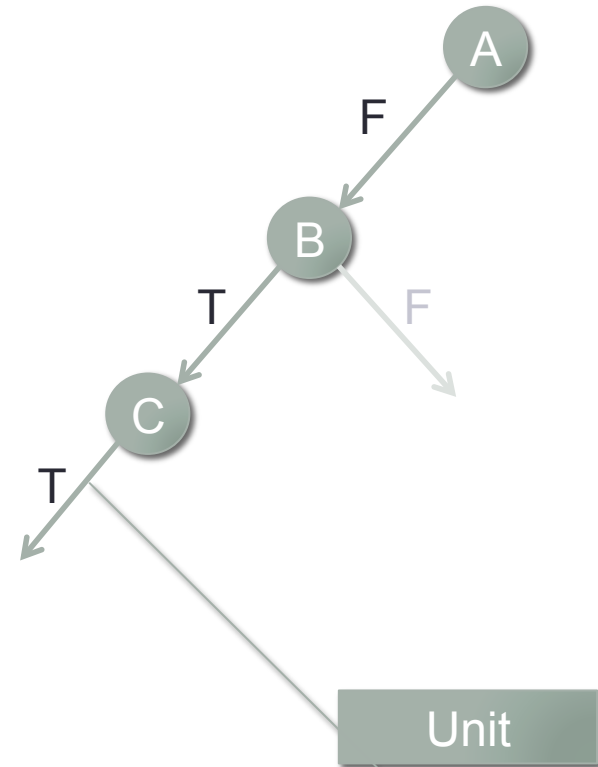
Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



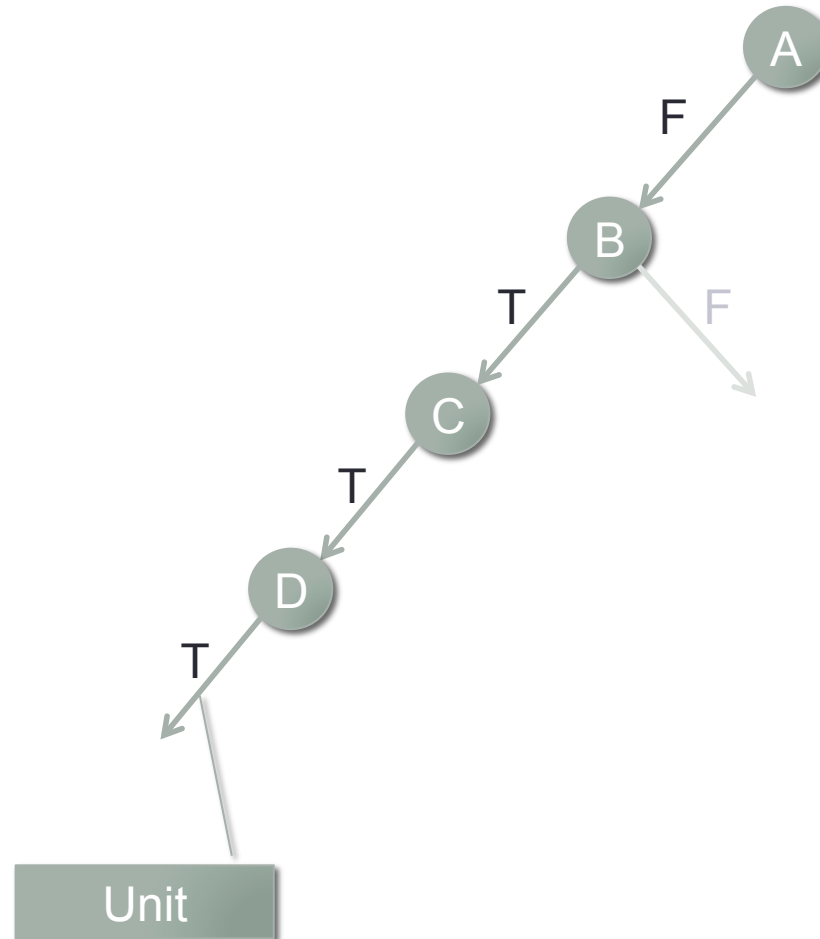
Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



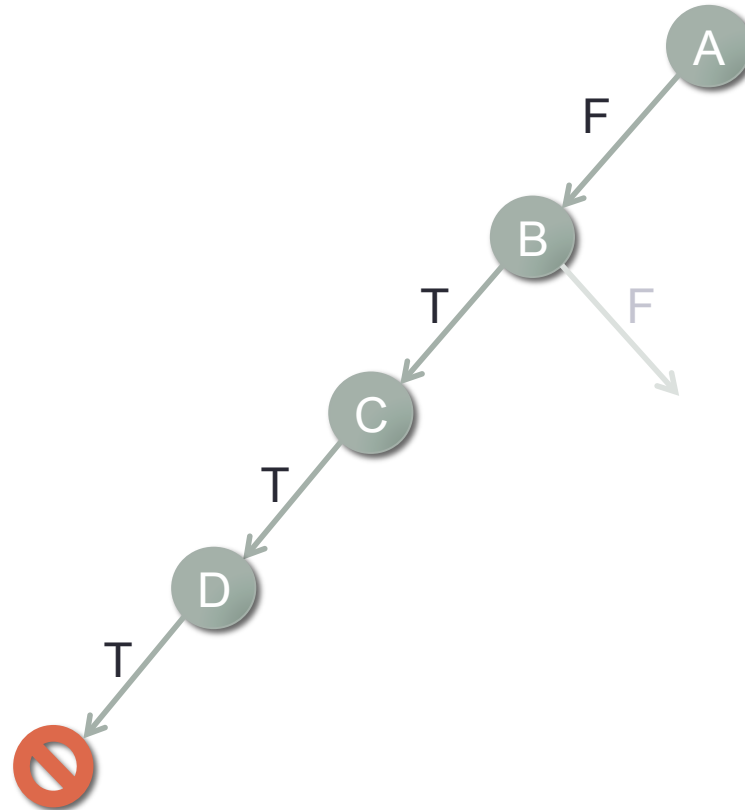
Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



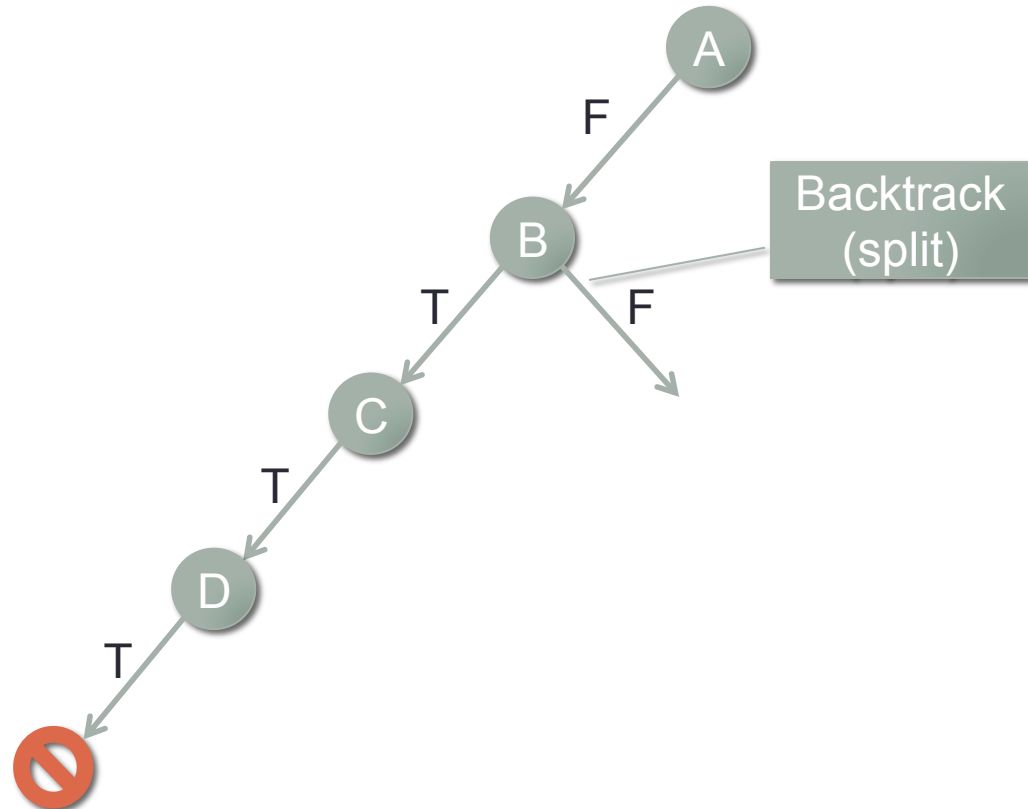
Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



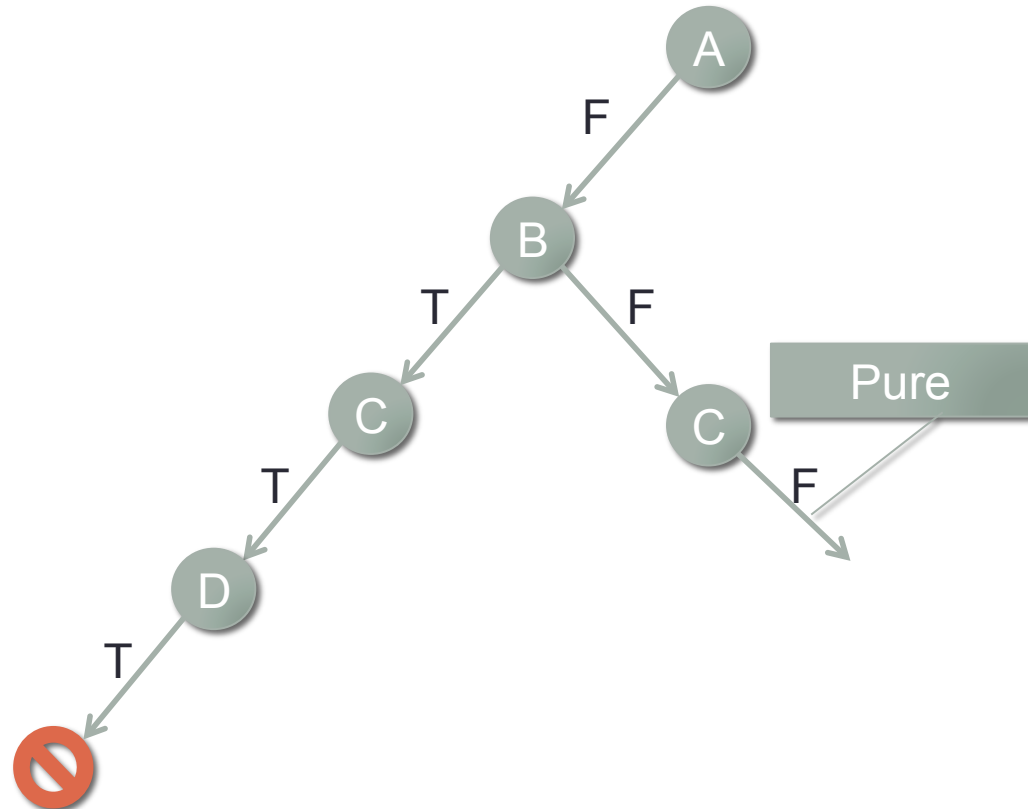
Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



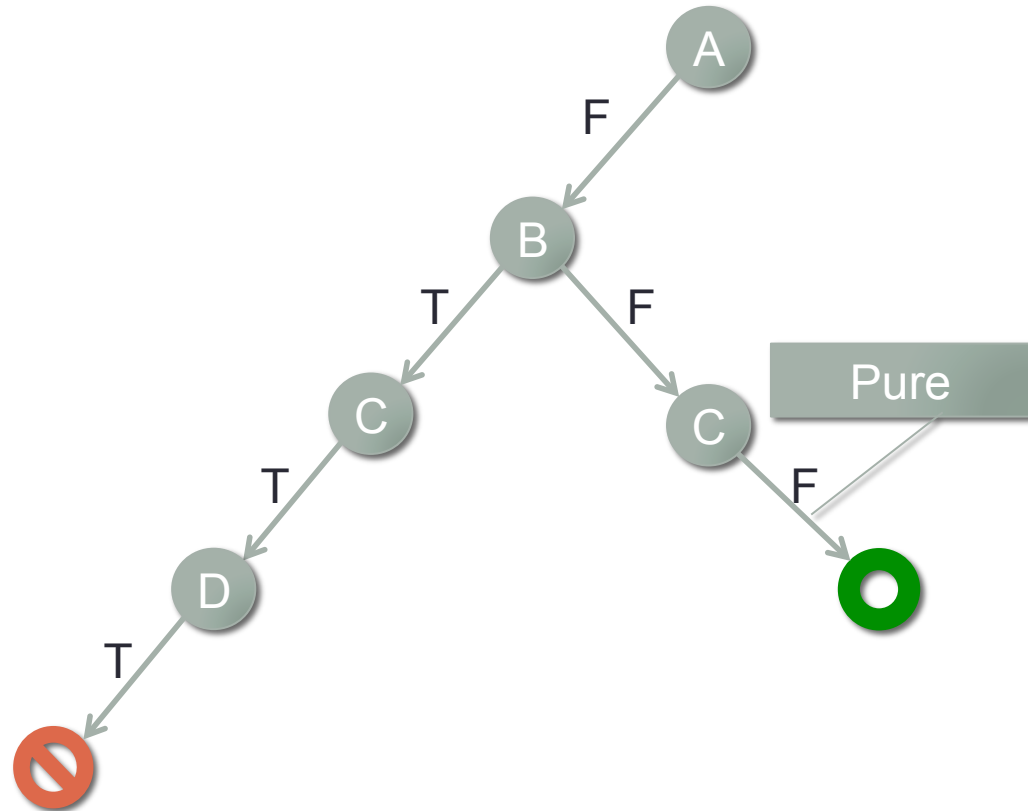
Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



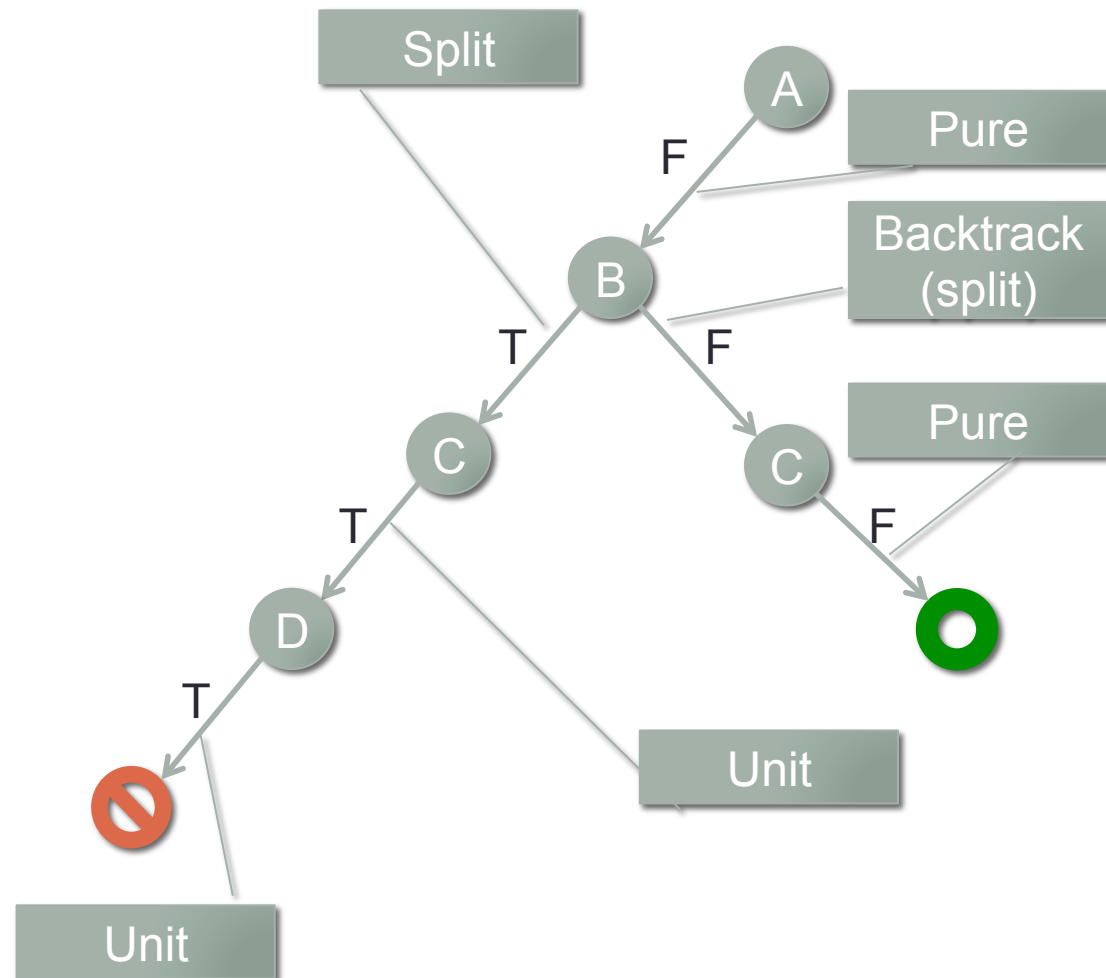
Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



Exemplo de aplicação

- $\neg A \vee B$
- $\neg A \vee \neg C \vee D$
- $B \vee \neg C$
- $\neg B \vee C$
- $\neg C \vee D$
- $\neg C \vee \neg D$



Propriedades do algoritmo DPLL

- Completo
- Eficaz na prática, podendo resolver problemas de verificação de Hardware com 1 milhão de variáveis
- O algoritmo DPLL é, na verdade, uma família de algoritmos, pois depende da:
 - Escolha do símbolo na heurística do símbolo puro
 - Escolha da cláusula na heurística da cláusula unitária
 - Escolha do símbolo no “split”
 - Ótimizações em cada passo

Resumo das classes de complexidade

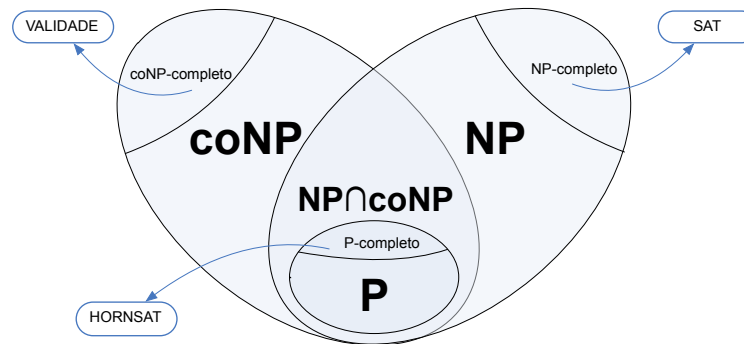
(P)

- Um problema pertence à classe P quando pode ser resolvido por uma máquina de Turing determinista em tempo polinomial.
- Um problema de decisão é P-completo se pertence a P e **qualquer** problema na classe P pode ser reduzido a ele (em espaço logarítmico).
- Saber se um conjunto de cláusulas de Horn é satisfatível é um problema P-completo (HORNSAT)
- Saber qual o valor de um circuito booleano dados os seus inputs é P-completo (CIRCUIT VALUE)
- Saber se um número inteiro é primo pertence a P (provado em 2002)!

Resumo das classes de complexidade (NP e coNP)

- Um problema pertence a classe **NP**, quando pode ser resolvido por uma máquina de Turing não determinista em tempo polinomial.
- Um problema de decisão é **NP-completo** quando a solução pode ser verificada em tempo polinomial. Formalmente, um problema é NP-completo
 1. se pertence a NP
 2. **qualquer** problema na classe NP pode ser reduzido a ele por uma transformação polinomial
- Caso um problema só obedeca ao critério (2) diz-se **NP-difícil**.
- Um problema pertence à classe **coNP** quando o seu complementar pertence à classe NP (pode-se verificar que não é solução em tempo polinomial).
- Presume-se que $P \neq NP$ e que $NP \neq coNP$.

Complexidade e Lógica Proposicional



- **Teorema de Cook-Levin (1971):** O problema SAT é NP-completo.
- **NOTA:** Testar a validade de um conjunto de fórmulas booleanas é um problema **coNP-completo** (VALIDITY).

Outros problemas NP-completos

- SAT
- 0-1 INTEGER PROGRAMMING
- CLIQUE
- SET PACKING
- VERTEX COVER
- SET COVERING
- FEEDBACK ARC SET
- FEEDBACK NODE SET
- DIRECTED HAMILTONIAN CIRCUIT
- UNDIRECTED HAMILTONIAN CIRCUIT
- 3-SAT
- CHROMATIC NUMBER
- CLIQUE COVER
- EXACT COVER
- 3D MATCHING
- STEINER TREE
- HITTING SET
- KNAPSACK
- JOB SEQUENCING
- PARTITION
- MAX-CUT

WalkSAT

- Trepa-colinas no espaço de atribuições completas
 - Algoritmo de pesquisa local.
- Em cada iteração, o algoritmo escolhe uma cláusula não satisfeita e um símbolo dessa cláusula para trocar. A forma de escolha do símbolo a trocar de valor é ela própria aleatória, podendo ser:
 - Utilizando a heurística “min-conflitos”, minimizando o número de cláusulas insatisfeitas no passo seguinte
 - Escolha aleatória do símbolo a trocar na cláusula (“passeio aleatório”)

WalkSAT

```
function WALKSAT(clauses, p, max-flips) returns a satisfying model or failure  
  inputs: clauses, a set of clauses in propositional logic  
           p, the probability of choosing to do a “random walk” move, typically  
           around 0.5  
           max-flips, number of flips allowed before giving up  
  model  $\leftarrow$  a random assignment of true/false to the symbols in clauses  
  for i = 1 to max-flips do  
    if model satisfies clauses then return model  
    clause  $\leftarrow$  a randomly selected clause from clauses that is false in model  
    with probability p flip the value in model of a randomly selected symbol  
    from clause  
    else flip whichever symbol in clause maximizes the number of satisfied clauses  
  return failure
```

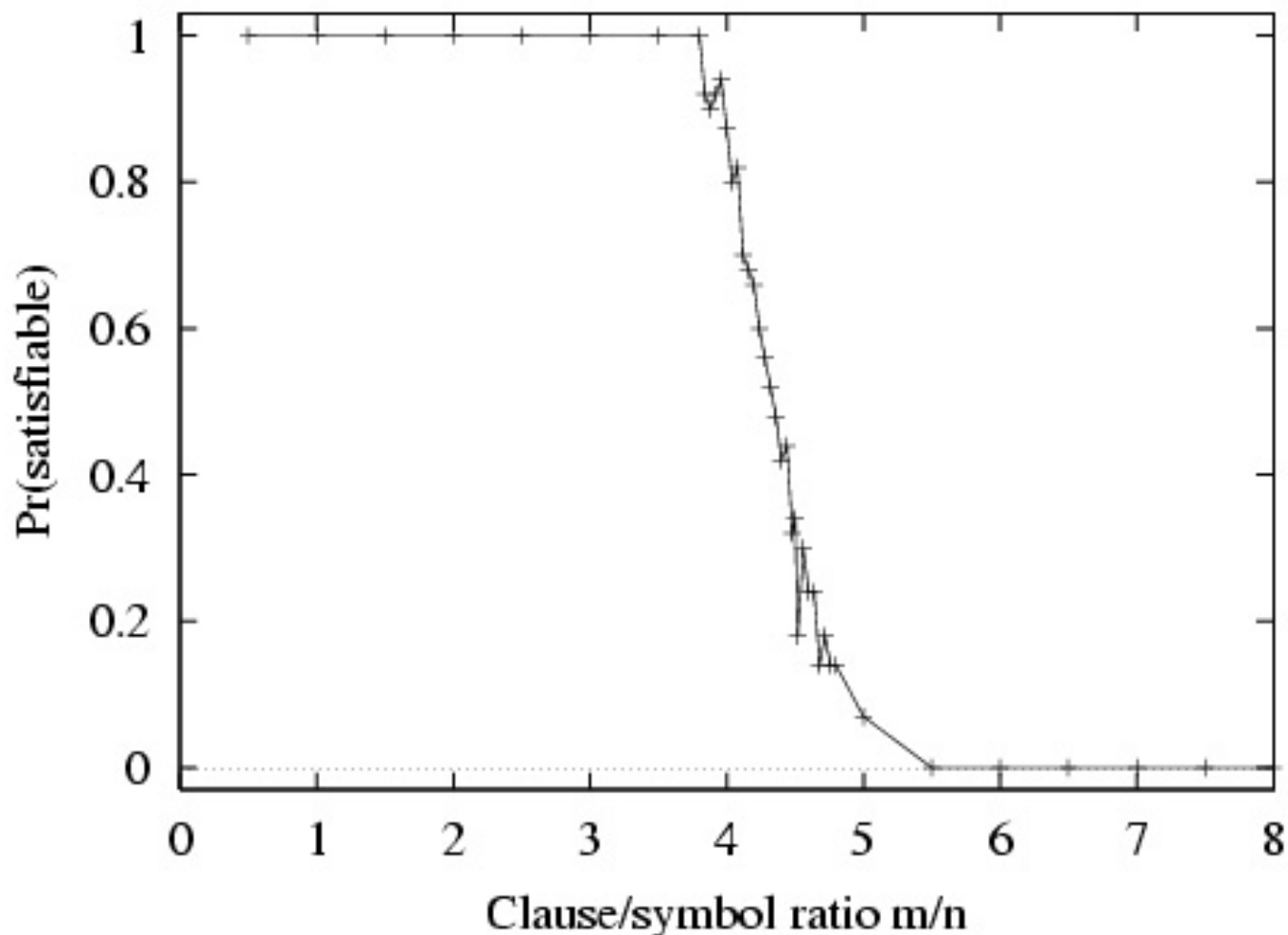
Propriedades do WalkSAT

- Incompleto
- Se uma proposição é insatisfazível então o algoritmo não termina: limita-se a max flips...
- Logo, procura local não serve em geral para resolver o problema da consequência lógica
- Algoritmos locais como o WalkSAT são mais eficazes quando se espera que uma solução exista
- Muito eficiente na prática...

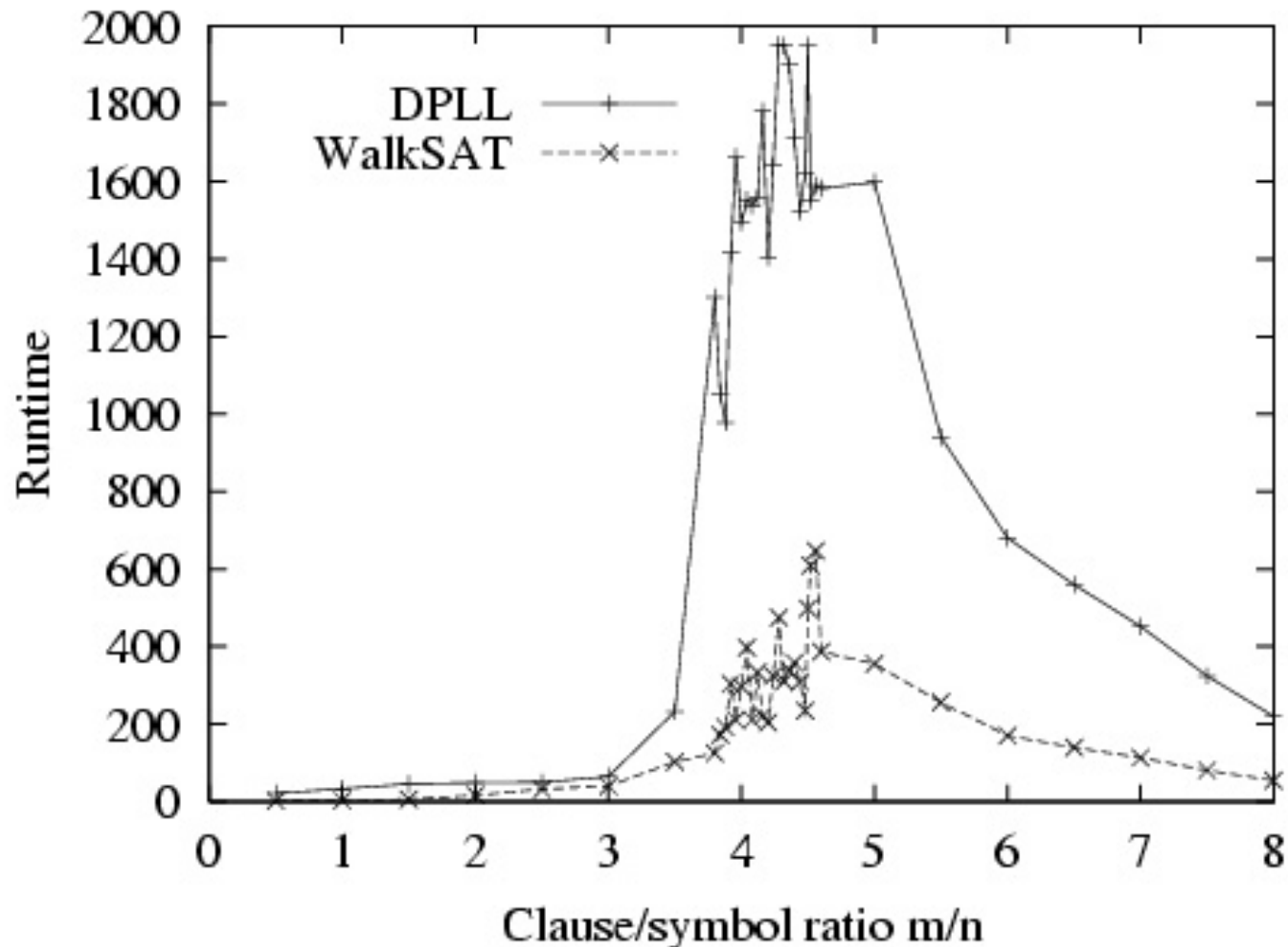
Problemas de satisfatibilidade difíceis

- Considere cláusulas 3-CNF geradas aleatoriamente, e.g:
 $(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee \neg E) \wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C)$
- Seja
- m = número de cláusulas
- n = número de símbolos
- Os problemas mais difíceis parecem concentrar-se perto do valor do rácio $m/n = 4.3$ (ponto crítico)

Problemas de satisfatibilidade difíceis



Problemas de satisfatibilidade difíceis



Um agente lógico no mundo do Wumpus

- Formulação em lógica proposicional:

$$\neg P_{1,1}$$

$$\neg W_{1,1}$$

$$B_{x,y} \Leftrightarrow (P_{x,y+1} \vee P_{x,y-1} \vee P_{x+1,y} \vee P_{x-1,y})$$

$$S_{x,y} \Leftrightarrow (W_{x,y+1} \vee W_{x,y-1} \vee W_{x+1,y} \vee W_{x-1,y})$$

$$W_{1,1} \vee W_{1,2} \vee \dots \vee W_{4,4}$$

$$\neg W_{1,1} \vee \neg W_{1,2}$$

$$\neg W_{1,1} \vee \neg W_{1,3}$$

$\vdots$

- São necessárias 64 variáveis proposicionais e 155 frases
- Uma casa na fronteira é demonstravelmente segura se a frase $(\neg P_{i,j} \wedge \neg W_{i,j})$ é uma consequência lógica da KB.

Um agente lógico no mundo do Wumpus

```
function PL-WUMPUS-AGENT(percept) returns an action
  inputs: percept, a list, [stench,breeze,glitter]
  static: KB, a knowledge base, initially containing the “physics” of the world
           x, y, orientation, the agent’s position (initially [1,1]) and orientation (initially right)
           visited, an array indicating which squares have been visited, initially false
           action, the agent’s most recent action, initially null
           plan, an action sequence, initially empty

  update x,y,orientation, visited based on action
  if stench then TELL(KB,  $S_{x,y}$ ) else TELL(KB,  $\neg S_{x,y}$ )
  if breeze then TELL(KB,  $B_{x,y}$ ) else TELL(KB,  $\neg B_{x,y}$ )
  if glitter then action  $\leftarrow$  grab
  else if plan is nonempty then action  $\leftarrow$  POP(plan)
  else if for some fringe square [i,j], ASK(KB, ( $\neg P_{i,j} \wedge \neg W_{i,j}$ )) is true or
           for some fringe square [i,j], ASK(KB, ( $P_{i,j} \vee W_{i,j}$ )) is false then do
           plan  $\leftarrow$  A*-GRAPH-SEARCH(ROUTE-PROBLEM([x,y], orientation, [i,j], visited))
           action  $\leftarrow$  POP(plan)
  else action  $\leftarrow$  a randomly chosen move
  return action
```

Limites da expressividade da lógica proposicional

- KB contém cláusulas capturando as leis “física” para cada casa
 - Seria melhor ter apenas duas frases, uma para aragens e uma para o mau cheiro, que fossem válidas para todas as casas.
- Para cada instante t e casa $[x,y]$,
$$L_{x,y} \wedge \text{FacingRight}^t \wedge \text{Forward}^t \Rightarrow L_{x+1,y}$$
- Proliferação rápida do número de cláusulas

O SuDoKu* em lógica proposicional

8								3
			5	9	1			4
	5	7		3		6		
	4		8		2		6	
		3		7	6		1	
	7	8				5	9	
2					9	8		
		1	6				3	
4				2				9

- Cada célula contém um inteiro entre 1 e 9
 - Nenhum par de células na mesma linha contém o mesmo valor
 - Nenhum par de células na mesma coluna contém o mesmo valor
 - Nenhum par de células num bloco 3x3 contém o mesmo valor
- Saber se existe ou não solução para um dado puzzle SuDoKu é um problema NP-completo e, como tal, pode ser reduzido por uma transformação polinomial a um problema de satisfatibilidade booleana. * - “suji wa dokushin ni kagiru”

Tradução do SuDoKu para lógica proposicional

8								3
			5	9	1			4
	5	7		3		6		
	4		8		2		6	
		3		7	6		1	
	7	8				5	9	
2					9	8		
		1	6				3	
4				2				9

- São necessárias $n \times n \times n = n^3$ variáveis para um SuDoKu $n \times n$.
- Variável $c_{i,j,k}$ ($1 \leq i, j, k \leq n$) é verdadeira quando $i \times j$ contém o valor k .
- **Cláusulas de célula** ($n^2 \times (1 + (n \times (n-1))/2)$)

$$c_{i,j,1} \vee \dots \vee c_{i,j,k}$$

uma cláusula por cada casa $1 \leq i, j \leq n$

$$\neg c_{i,j,k} \vee \neg c_{i,j,l} \quad (1 \leq k < l \leq n)$$

$n \times (n-1)/2$ cláusulas para cada casa $1 \leq i, j \leq n$

- **Cláusulas de linha** ($n^2 \times (1 + (n \times (n-1))/2)$)

$$c_{i,1,k} \vee \dots \vee c_{i,n,k}$$

uma cláusula por cada casa $1 \leq i, k \leq n$

$$\neg c_{i,j,k} \vee \neg c_{i,l,k} \quad (1 \leq j < l \leq n)$$

$n \times (n-1)/2$ cláusulas para cada casa $1 \leq i, k \leq n$

- Adicionam-se cláusulas semelhantes para tratar as colunas e os blocos, obtendo
- um total $2n^4 - 2n^3 + 4n^2$ cláusulas.
 - Para um puzzle 9x9 temos assim exactamente 11988 cláusulas.
- Junta-se a proposição $c_{i,j,k}$ para cada casa i, j ocupada com o valor k .

O Nosso puzzle SuDoKu em lógica proposicional

8	1	4	2	6	7	9	5	3
3	6	2	5	9	1	7	8	4
9	5	7	4	3	8	6	2	1
1	4	9	8	5	2	3	6	7
5	2	3	9	7	6	4	1	8
6	7	8	1	4	3	5	9	2
2	3	5	7	1	9	8	4	6
7	9	1	6	8	4	2	3	5
4	8	6	3	2	5	1	7	9

$c_{1,1,1} \vee c_{1,1,2} \vee \dots \vee c_{1,1,8} \vee c_{1,1,9}$ (células)

$\neg c_{1,1,1} \vee \neg c_{1,1,2}$

$\neg c_{1,1,1} \vee \neg c_{1,1,3}$

$\vdots$

$c_{1,1,1} \vee c_{1,2,1} \vee \dots \vee c_{1,8,1} \vee c_{1,9,1}$ (linhas)

$\neg c_{1,1,1} \vee \neg c_{1,2,1}$

$\neg c_{1,1,1} \vee \neg c_{1,3,1}$

$\vdots$

$c_{1,1,1} \vee c_{2,1,1} \vee \dots \vee c_{8,1,1} \vee c_{9,1,1}$ (colunas)

$\neg c_{1,1,1} \vee \neg c_{2,1,1}$

$\neg c_{1,1,1} \vee \neg c_{3,1,1}$

$\vdots$

$c_{1,1,1} \vee c_{1,2,1} \vee \dots \vee c_{3,2,1} \vee c_{3,3,1}$ (blocos)

$\neg c_{1,1,1} \vee \neg c_{2,2,1}$

$\neg c_{1,1,1} \vee \neg c_{2,3,1}$

$\vdots$

$c_{1,1,8} \wedge c_{1,9,3} \wedge \dots \wedge c_{9,9,9}$ (valores)

- A solução do puzzle SuDoKu pode ser extraída das variáveis verdadeiras num modelo que satisfaça todas as cláusulas. Encontrar esse modelo é um problema da classe FNP (function NP problem, na terminologia inglesa).

Sumário

- Agentes lógicos aplicam inferência a bases de conhecimento para derivar nova informação
- e tomar decisões
- Conceitos básicos de lógica:
 - sintaxe: estrutura formal das frases declarativas
 - semântica: veracidade das frases relativamente a modelos
 - conclusão: verdade necessária de uma frase dado outra
 - inferência: derivação de frases a partir de outras frases
 - solidez: derivações produzem apenas frases que são conclusões lógicas
 - completude: derivações conseguem produzir todas as frases que são consequência lógicas
- O mundo do Wumpus requer a capacidade de lidar com informação parcial e negativa, raciocínio por casos, etc.
- Encadeamento para a frente e para trás têm complexidade temporal linear na dimensão da KB, completos para cláusulas de Horn. Resolução é completa para a lógica proposicional
- A lógica proposicional não tem poder expressivo suficiente