# Diagramas de Decisão Binária (BDDs)

## Aula 2

Luiz Carlos Vieira

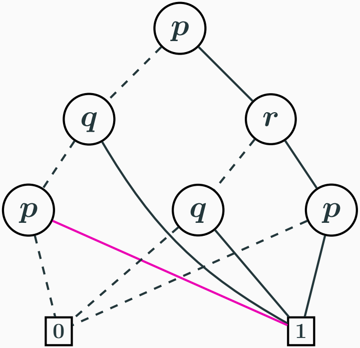
7 de outubro de 2015

MAC0239 - Introdução à Lógica e Verificação de Programas

# Conteúdo

- BDDs ordenados e reduzidos (ROBDDs)
- Algoritmos para ROBDDs
    - algoritmo `reduzir`
    - algoritmo `aplicar`
    - algoritmo `restringir`
    - algoritmo `existe`

# Relembrando: múltiplas ocorrências

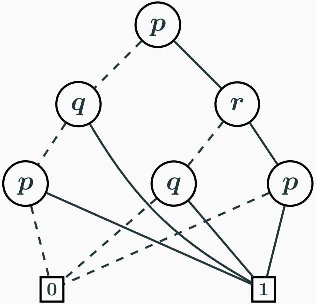


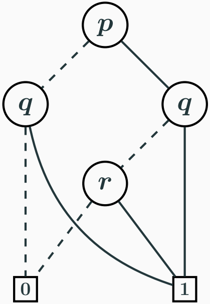

- A definição de BDDs não impede uma variável de ocorrer mais de uma vez em um caminho
- Mas tal representação pode incorrer em desperdícios
  - linha sólida do $p$ à esquerda (colorida) jamais será percorrida

Esse é um resultado comum após as operações discutidas na aula anterior

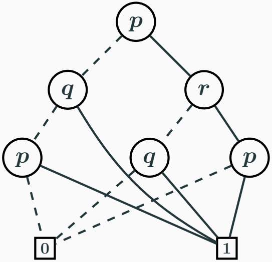
# Relembrando: comparação de BDDs

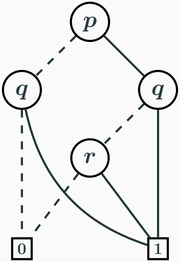
Além de tornar um BDD menos eficiente, ocorrências múltiplas de uma variável também dificultam a comparação de BDDs



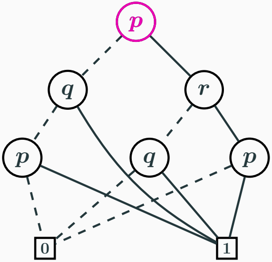
# Conceito de ordem de um caminho

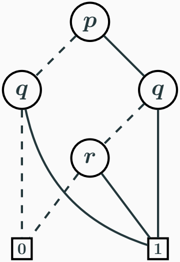


# Conceito de ordem de um caminho

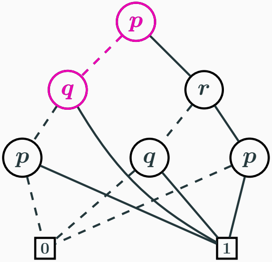
- $[p \qquad ]$

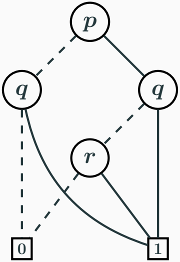


# Conceito de ordem de um caminho

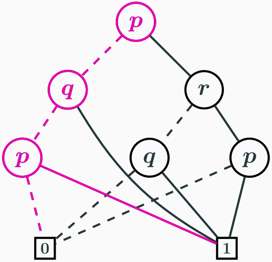
- $[p, q \quad ]$

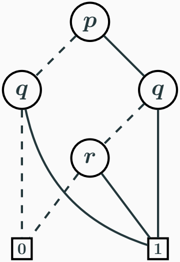


# Conceito de ordem de um caminho

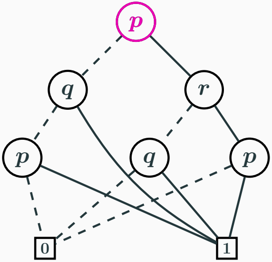
- $[p, q, p]$

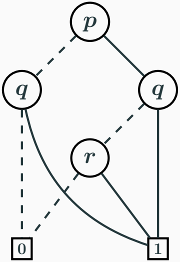


# Conceito de ordem de um caminho

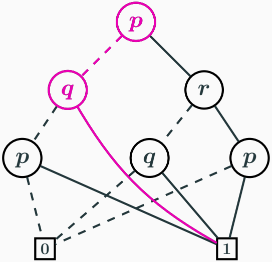
- $[p, q, p]$
- $[p \quad ]$

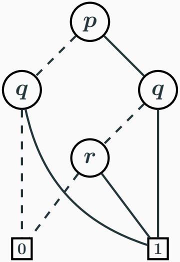


# Conceito de ordem de um caminho

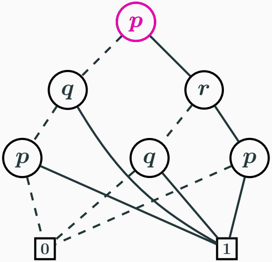
- $[p, q, p]$
- $[p, q]$

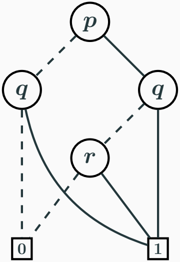


# Conceito de ordem de um caminho

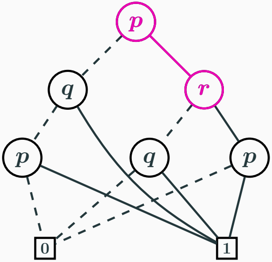
- $[p, q, p]$
- $[p, q]$
- $[p \qquad ]$

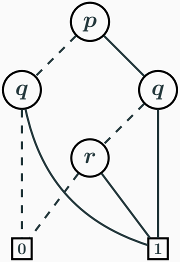


# Conceito de ordem de um caminho

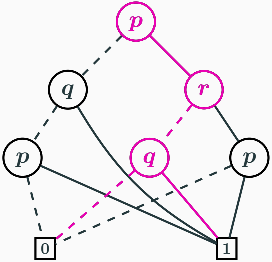
- $[p, q, p]$
- $[p, q]$
- $[p, r \quad ]$

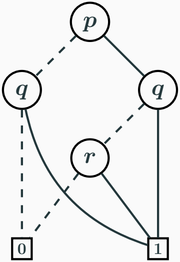


# Conceito de ordem de um caminho

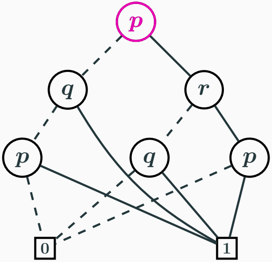
- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$

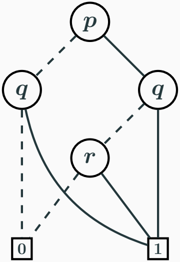


# Conceito de ordem de um caminho

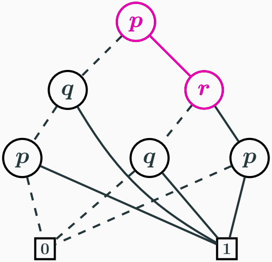
- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p \qquad ]$

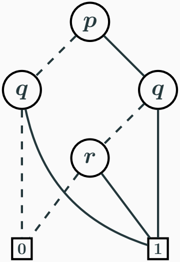


# Conceito de ordem de um caminho

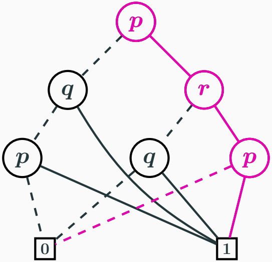
- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r \quad ]$

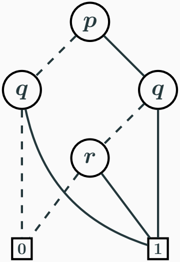


# Conceito de ordem de um caminho

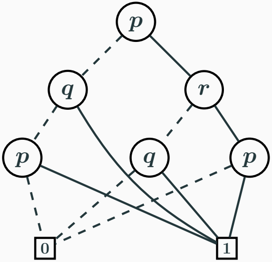
- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$



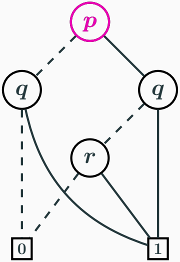



# Conceito de ordem de um caminho

- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$

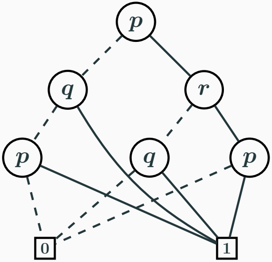



- $[p \quad ]$

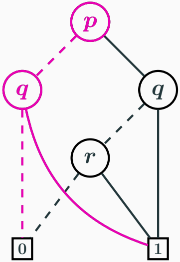


# Conceito de ordem de um caminho

- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$

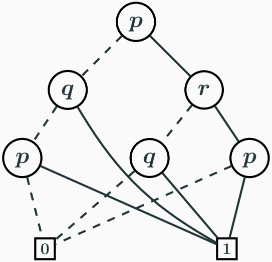



- $[p, q]$

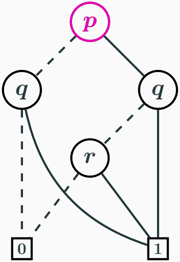


# Conceito de ordem de um caminho

- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$

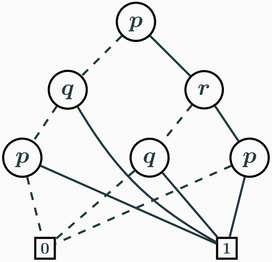



- $[p, q]$
- $[p \quad\quad ]$

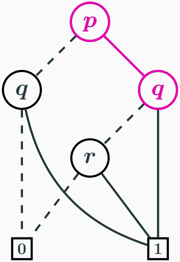


# Conceito de ordem de um caminho

- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$

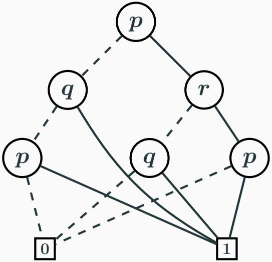



- $[p, q]$
- $[p, q \quad ]$

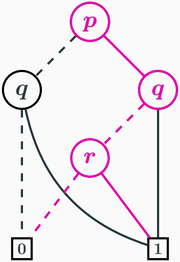


# Conceito de ordem de um caminho

- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$

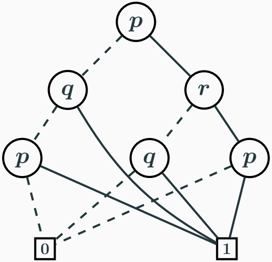



- $[p, q]$
- $[p, q, r]$

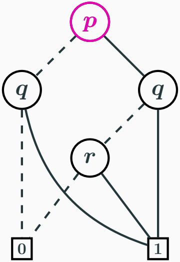


# Conceito de ordem de um caminho

- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$

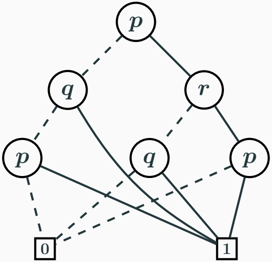



- $[p, q]$
- $[p, q, r]$
- $[p \quad ]$

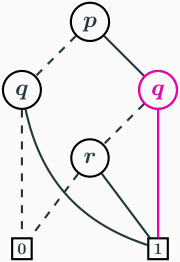


# Conceito de ordem de um caminho

- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$

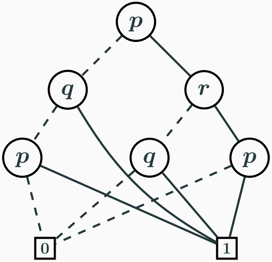



- $[p, q]$
- $[p, q, r]$
- $[p, q]$

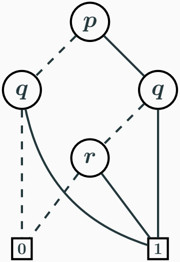


# Conceito de ordem de um caminho

- $[p, q, p]$
- $[p, q]$
- $[p, r, q]$
- $[p, r, p]$




- $[p, q]$
- $[p, q, r]$
- $[p, q]$



# BDDs ordenados

Quando a ordem das variáveis em qualquer caminho é sempre a mesma, o BDD passa a ser chamado Diagrama de Busca Binária Ordenado (OBDD)

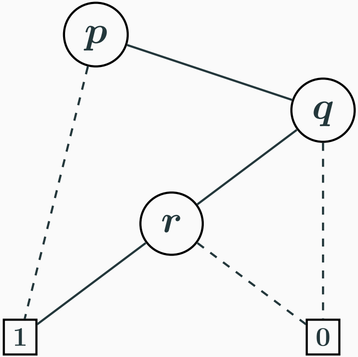
# Definição: OBDDs

## Definição 6.6

Seja $[p_1, p_2, ..., p_n]$ uma lista ordenada de variáveis sem duplicação e seja $B$ um BDD tal que todas as suas variáveis aparecem em algum lugar da lista. Dizemos que $B$ tem a ordem $[p_1, p_2, ..., p_n]$ se todos os nós de variáveis de $B$ ocorrem na lista, e, para toda ocorrência de $p_i$ seguido de $p_j$ ao longo de qualquer caminho em $B$ (ou seja, $p_i \prec p_j$), temos $i < j$.

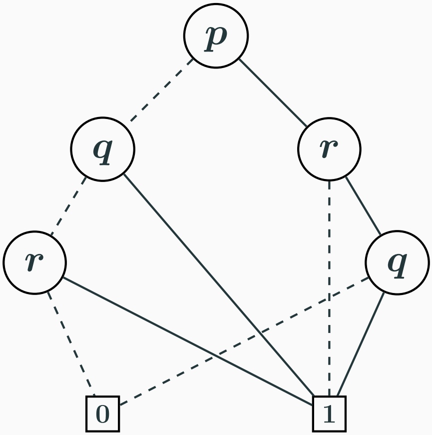
# Exemplo de BDD ordenado

Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}]$ (em qualquer caminho)



# Exemplo de BDD não ordenado

Sem ordem única ($[p, q, r]$ à esquerda e $[p, r, q]$ à direita)



# OBDDs reduzidos

Quando são reduzidos, OBDDs passam a ser chamados de Diagramas de Busca Binária Ordenados Reduzidos (ROBDD)

# Vantagens da ordenação de BDDs

- Reduções em um OBDDs mantêm ordem original
  - C1: compartilha nós terminais
  - C2: elimina nós não-terminais redundantes
  - C2: compartilha sub-diagramas idênticos
- Compromisso com ordem e redução produzem representação única de funções booleanas
  - chamada de *forma canônica*
- Comparação de ROBDDs de ordens compatíveis é imediata
  - basta verificar se suas estruturas são idênticas

# Teorema: ROBDDs são únicos

## Teorema 6.7

A representação em ROBDD de uma função dada $\phi$ é unica. Isto é, sejam $\boldsymbol{B}$ e $\boldsymbol{B'}$ dois ROBDDs com ordens compatíveis; se $\boldsymbol{B}$ e $\boldsymbol{B'}$ representam a mesma função booleana, então eles têm estruturas idênticas.

# Consequências da forma canônica

- **Teste de equivalência semântica**. Se duas funções são representadas por OBDDs com ordens compatíveis, é possível decidir eficientemente se são equivalentes reduzindo-os e comparando sua estrutura;

# Consequências da forma canônica

- **Teste de equivalência semântica**. Se duas funções são representadas por OBDDs com ordens compatíveis, é possível decidir eficientemente se são equivalentes reduzindo-os e comparando sua estrutura;

- **Ausência de variáveis redundantes**. Se o valor de uma função booleana não depende de uma variável, então o ROBDD que a representa não contém tal variável;

# Consequências da forma canônica

- **Teste de equivalência semântica**. Se duas funções são representadas por OBDDs com ordens compatíveis, é possível decidir eficientemente se são equivalentes reduzindo-os e comparando sua estrutura;
- **Ausência de variáveis redundantes**. Se o valor de uma função booleana não depende de uma variável, então o ROBDD que a representa não contém tal variável;
- **Teste de validade**. Se uma função booleana é válida, seu ROBDD é igual a $\boldsymbol{B_1}$;

# Consequências da forma canônica

- **Teste de equivalência semântica**. Se duas funções são representadas por OBDDs com ordens compatíveis, é possível decidir eficientemente se são equivalentes reduzindo-os e comparando sua estrutura;
- **Ausência de variáveis redundantes**. Se o valor de uma função booleana não depende de uma variável, então o ROBDD que a representa não contém tal variável;
- **Teste de validade**. Se uma função booleana é válida, seu ROBDD é igual a $\boldsymbol{B_1}$;
- **Teste de satisfação**. Se uma função booleana é satisfeita, então seu ROBDD não é igual a $\boldsymbol{B_0}$;

# Consequências da forma canônica

- **Teste de equivalência semântica**. Se duas funções são representadas por OBDDs com ordens compatíveis, é possível decidir eficientemente se são equivalentes reduzindo-os e comparando sua estrutura;
- **Ausência de variáveis redundantes**. Se o valor de uma função booleana não depende de uma variável, então o ROBDD que a representa não contém tal variável;
- **Teste de validade**. Se uma função booleana é válida, seu ROBDD é igual a $\boldsymbol{B_1}$;
- **Teste de satisfação**. Se uma função booleana é satisfeita, então seu ROBDD não é igual a $\boldsymbol{B_0}$;
- **Teste de implicação**. Pode-se testar se uma função $\phi$ implica em outra $\psi$ calculando o ROBDD para $\phi \wedge \neg\psi$; a implicação é verdadeira se e somente se este ROBDD é igual a $\boldsymbol{B_0}$.

# Impacto da escolha da ordenação

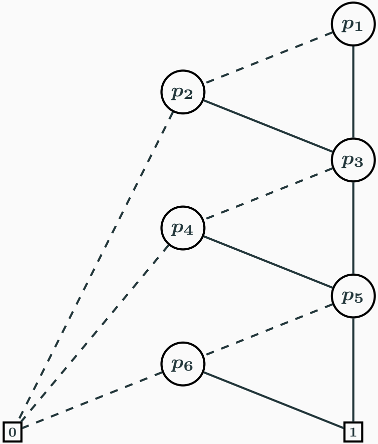
Considere a escolha da ordem de variáveis para a seguinte função booleana em CNF:
$\phi \equiv (p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge ... \wedge (p_{2n-1} \vee p_{2n})$

- Se a escolha for a "*ordem natural de ocorrência na fórmula*" ($[p_1, p_2, p_3, ..., , p_{2n-1}, p_{2n}]$), o ROBDD terá $2n + 2$ nós

- Se a escolha for "*índices impares antes de índices pares*" ($[p_1, p_3, p_5, ..., p_{2n-1}, p_2, p_4, p_6, ..., p_{2n}]$), o ROBDD terá $2^{n+1}$ nós

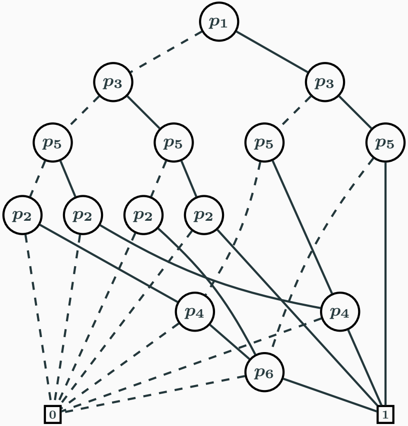
# Ordem "natural" para $n = 3$

ROBDD para $\phi \equiv (p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge (p_5 \wedge p_6)$ com a ordem de variáveis $[p_1, p_2, p_3, p_4, p_5, p_6]$



# Ordem "ímpares $\prec$ pares" para $n = 3$

ROBDD para $\phi \equiv (p_1 \vee p_2) \wedge (p_3 \vee p_4) \wedge (p_5 \wedge p_6)$ com a ordem de variáveis $[p_1, p_3, p_5, p_2, p_4, p_6]$



# Escolha da ordenação

- A sensibilidade do tamanho de um ROBDD à ordem escolhida é o preço pago pelas facilidades obtidas

- Encontrar a ordem ótima também é um problema computacional caro
  - mas há heurísticas para ordens razoavelmente boas
  - tipicamente, agrupa-se as variáveis com interações mais fortes

# ROBDDs como representação

- RODDBs permitem representações compactas de certas classes de funções booleanas
  - que seriam exponenciais em outros formatos/representações
- Por outro lado, não se pode realizar as operações $\wedge$ e $\vee$ da forma "inocente" anteriormente estudada
  - elas podem introduzir ocorrências múltiplas de variáveis

# Principais algoritmos para ROBDDs

As seguintes operações estão no cerne do uso sério de ROBDDs:

- **Redução**. Permite reduzir OBDDs de forma eficiente
  - consiste basicamente da aplicação das simplificações C1-C3
- **Aplicação**. Permite realizar as operações lógicas $\wedge$, $\vee$ e $\neg$ (via $\phi \oplus \mathbf{1}$)
  - mantendo o BDD ordenado e reduzido

# Estrutura de dados

Os algoritmos das operações com ROBDDs utilizam como estrutura de dados uma tabela $\boldsymbol{T : \langle v, i_l, i_h \rangle \mapsto i_v}$, tal que:

- $\boldsymbol{\langle v, i_l, i_r \rangle}$ representa um nó qualquer no ROBDD
  - com uma variável $\boldsymbol{v}$
  - e identificadores de seus nós-filhos $\boldsymbol{i_l}$ (*low*, pela linha pontilhada) e $\boldsymbol{i_h}$ (*high*, pela linha sólida)

- $\boldsymbol{i_v}$ representa um inteiro positivo que serve como identificador único do nó da variável $\boldsymbol{v}$

# Ilustração dessa estrutura de dados

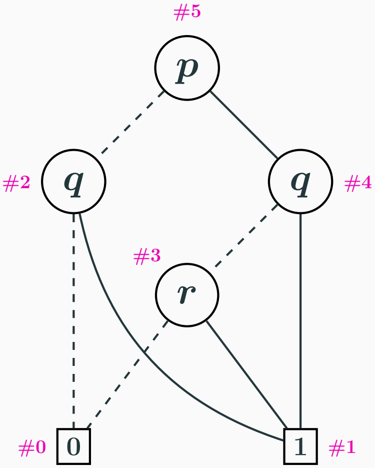



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |
| $\langle r, 0, 1 \rangle$ | $3$ |
| $\langle q, 3, 1 \rangle$ | $4$ |
| $\langle p, 2, 4 \rangle$ | $5$ |

# Observações

Nos algoritmos apresentados a seguir, assume-se que:

- A tabela $\boldsymbol{T}$ é uma variável global e $|\boldsymbol{T}|$ é o número de linhas existentes nessa tabela
- $\boldsymbol{T}(\langle \boldsymbol{v}, \boldsymbol{i_l}, \boldsymbol{i_h} \rangle) = \textsc{null}$ quando $(\boldsymbol{i_v}, \langle \boldsymbol{v}, \boldsymbol{i_l}, \boldsymbol{i_h} \rangle) \notin \boldsymbol{T}$
- LO e HI acessam os nós-filhos de um nó
- ID acessa o identificador de um nó
- VAR acessa a variável de um nó

# Algoritmo de redução

Reduz um OBDD de maneira recorrente. Funciona assim:

1. Percorre o OBDD de baixo para cima (busca em profundidade) atribuindo identificadores inteiros aos nós
2. Se o nó for terminal, atribui ou reutiliza o identificador (simplificação C1)
3. Se os identificadores dos filhos forem iguais, atribui ao nó esse mesmo identificador (simplificação C2)
4. Se existir outro nó com os mesmos filhos, atribui seu identificador ao nó (simplificação C3)
5. Caso contrário, atribui ao nó o próximo inteiro livre

# Pseudocódigo de GETNODE

**precondição:** recebe uma variável e os identificadores dos nós canônicos de seus filhos
**pós-condição:** devolve o identificador do nó canônico da variável

```
função GETNODE(v, i_l, i_h)
    se v ∉ {0, 1} então
        se i_l = i_h então                              ▷ simplificação C2
            devolva i_l
    i ← T(⟨v, i_l, i_h⟩)
    se i = NULL então                                   ▷ simplificações C1/C3
        i = |T|
        T ← T ∪ {(i, ⟨v, i_l, i_h⟩)}
    devolva i
```

# Pseudocódigo de REDUCE

**precondição:** recebe o nó raiz de um diagrama a ser reduzido
**pós-condição:** devolve o identificador do nó canônico (isto é, do diagrama reduzido)

**função** REDUCE$(n)$
    **se** $n \in \{0, 1\}$ **então** ▷ simplificação C1
        **devolva** GETNODE(VAR(n),NULL,NULL)
    $i_n \leftarrow T(\langle \text{VAR}(n), \text{ID}(\text{LO}(n)), \text{ID}(\text{HI}(n))\rangle)$
    **se** $i_n = \text{NULL}$ **então**
        $i_l \leftarrow \text{REDUCE}(\text{LO}(n))$
        $i_h \leftarrow \text{REDUCE}(\text{HI}(n))$
        $i_n \leftarrow \text{GETNODE}(\text{VAR}(n), i_l, i_h)$ ▷ simplificação C3
    **devolva** $i_n$

# Ilustração do algoritmo de redução

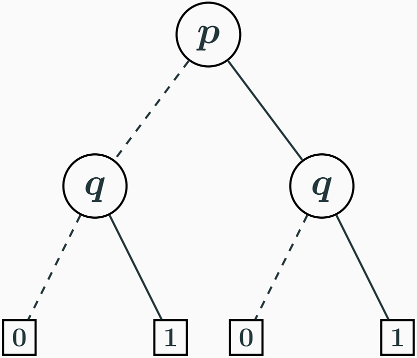



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|

# Ilustração do algoritmo de redução

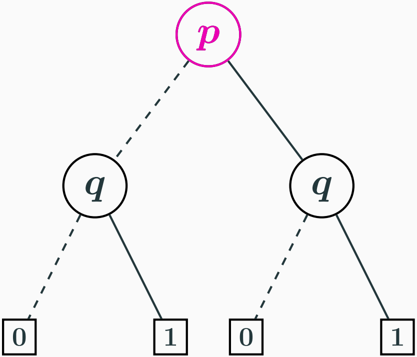



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|

# Ilustração do algoritmo de redução

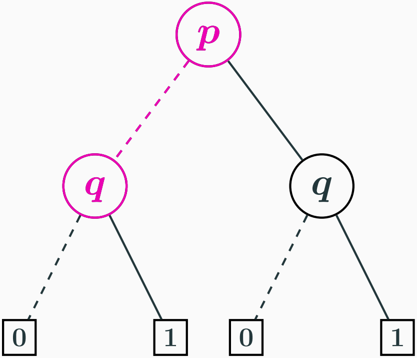



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|

# Ilustração do algoritmo de redução

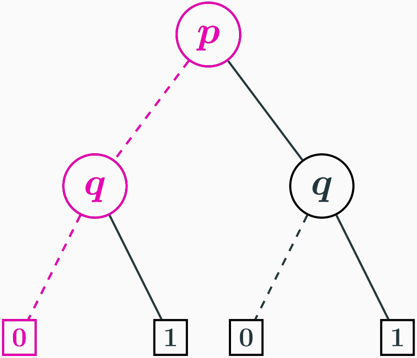



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|

# Ilustração do algoritmo de redução

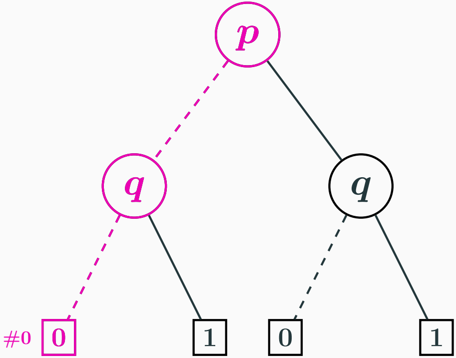



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |

# Ilustração do algoritmo de redução

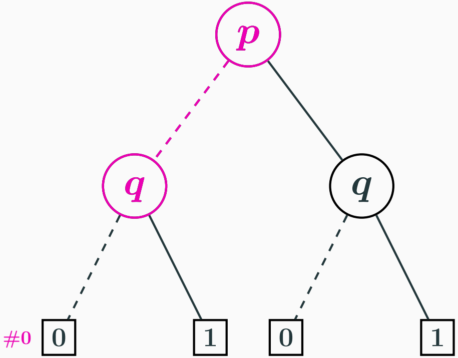



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |

# Ilustração do algoritmo de redução

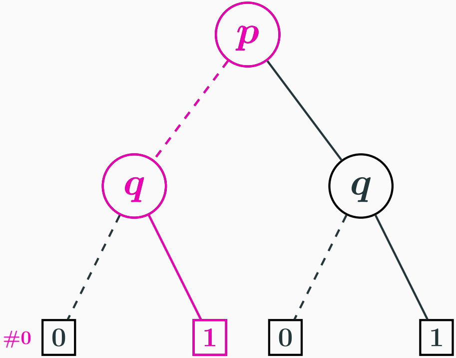



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |

# Ilustração do algoritmo de redução

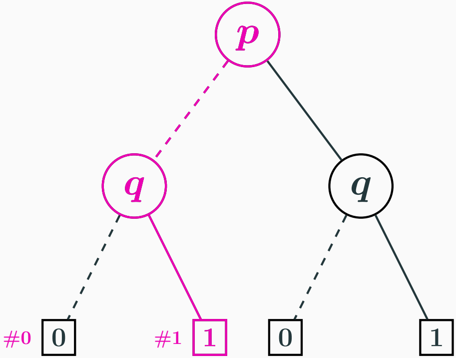



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |

# Ilustração do algoritmo de redução

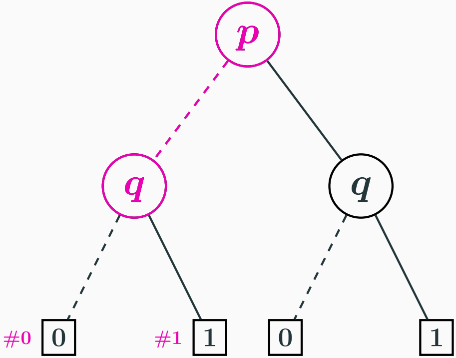



Tabela $\boldsymbol{T} : \langle \boldsymbol{v}, \boldsymbol{i_l}, \boldsymbol{i_h} \rangle \mapsto \boldsymbol{i_v}$

| $\boldsymbol{n}$ | $\boldsymbol{T(n)}$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |
| $\langle \mathbf{1}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{1}$ |

# Ilustração do algoritmo de redução

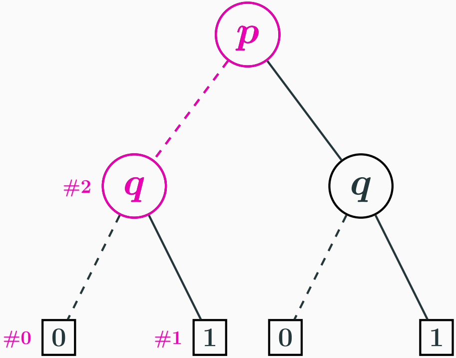



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |

# Ilustração do algoritmo de redução

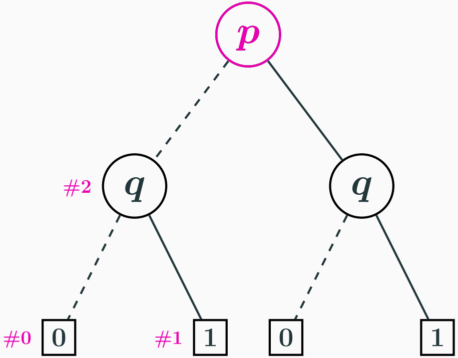



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |

# Ilustração do algoritmo de redução

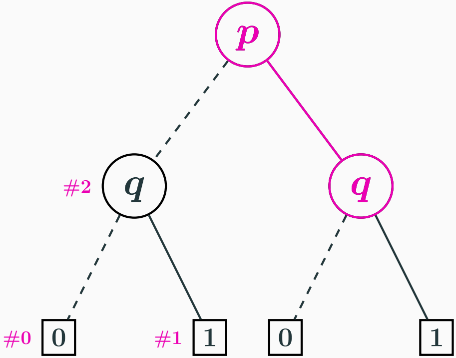



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |

# Ilustração do algoritmo de redução

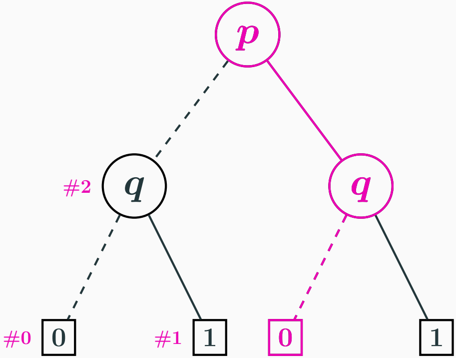



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |

# Ilustração do algoritmo de redução

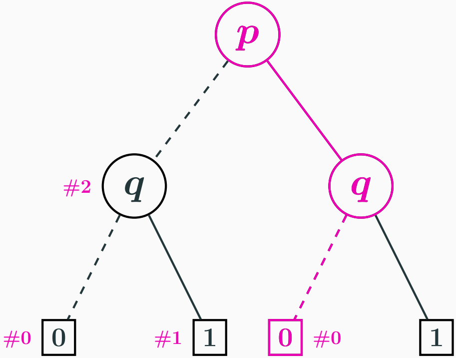



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |

# Ilustração do algoritmo de redução

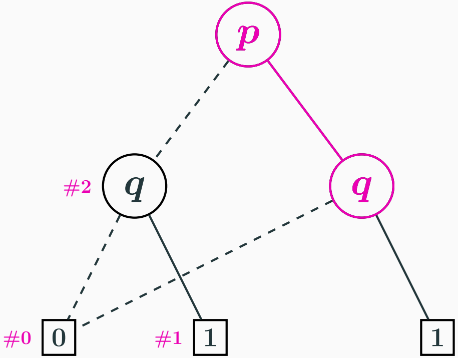



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |
| $\langle \mathbf{1}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{1}$ |
| $\langle q, 0, 1 \rangle$ | $\mathbf{2}$ |

# Ilustração do algoritmo de redução

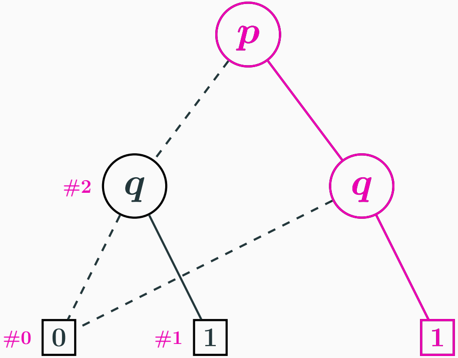



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |
| $\langle \mathbf{1}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{1}$ |
| $\langle q, 0, 1 \rangle$ | $\mathbf{2}$ |

# Ilustração do algoritmo de redução

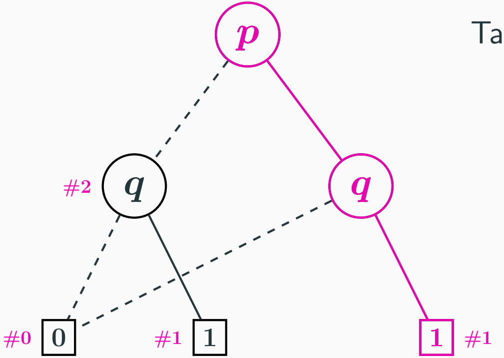



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |
| $\langle \mathbf{1}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{1}$ |
| $\langle q, 0, 1 \rangle$ | $\mathbf{2}$ |

# Ilustração do algoritmo de redução

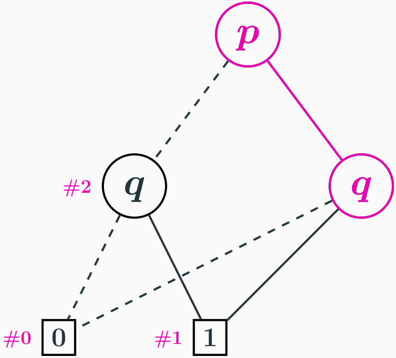



Tabela $\boldsymbol{T} : \langle \boldsymbol{v}, \boldsymbol{i_l}, \boldsymbol{i_h} \rangle \mapsto \boldsymbol{i_v}$

| $\boldsymbol{n}$ | $\boldsymbol{T(n)}$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |
| $\langle \mathbf{1}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{1}$ |
| $\langle \boldsymbol{q}, \mathbf{0}, \mathbf{1} \rangle$ | $\mathbf{2}$ |

# Ilustração do algoritmo de redução

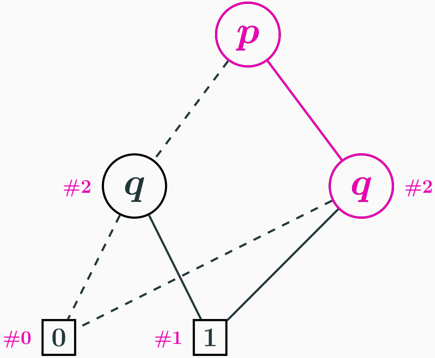



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle \mathbf{0}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{0}$ |
| $\langle \mathbf{1}, \text{NULL}, \text{NULL} \rangle$ | $\mathbf{1}$ |
| $\langle q, 0, 1 \rangle$ | $\mathbf{2}$ |

# Ilustração do algoritmo de redução

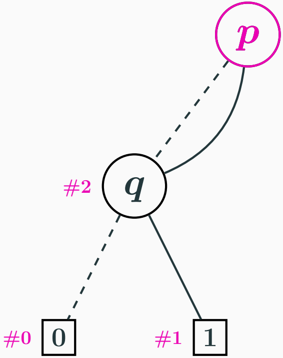



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |

# Ilustração do algoritmo de redução

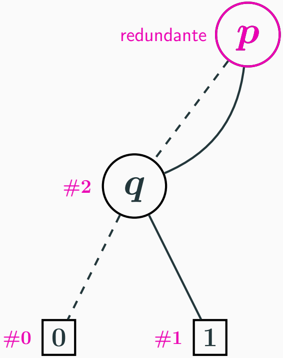



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |

# Ilustração do algoritmo de redução

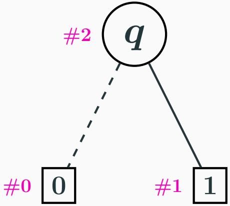



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle q, 0, 1 \rangle$ | $2$ |

# O compartilhamento é grande vantagem

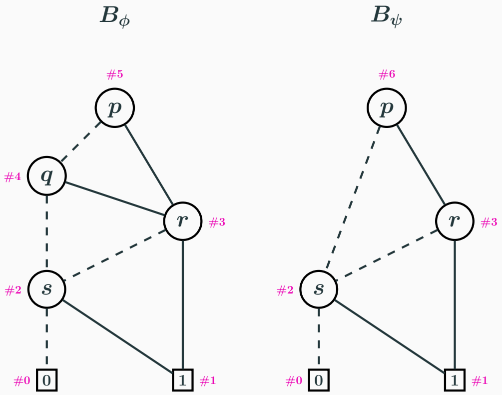



Tabela $T : \langle v, i_l, i_h \rangle \mapsto i_v$

| $n$ | $T(n)$ |
|---|---|
| $\langle 0, \text{NULL}, \text{NULL} \rangle$ | $0$ |
| $\langle 1, \text{NULL}, \text{NULL} \rangle$ | $1$ |
| $\langle s, 0, 1 \rangle$ | $2$ |
| $\langle r, 2, 1 \rangle$ | $3$ |
| $\langle q, 2, 3 \rangle$ | $4$ |
| $\langle p, 4, 3 \rangle$ | $5$ |
| $\langle p, 2, 3 \rangle$ | $6$ |

# Algoritmo de aplicação

Obtém o resultado de $\boldsymbol{B_\phi \; op \; B_\psi}$, sendo $\boldsymbol{op}$ uma operação booleana ($\wedge$, $\vee$, $\oplus$ ou $\neg$ via $\boldsymbol{B_\phi \oplus 1}$). Funciona assim:

1. inicia com a variável $\boldsymbol{v}$ de maior ordem (mais à esquerda na lista de ordenação)
2. divide o problema em dois subproblemas, dependendo de $\boldsymbol{v}$ ser $\mathbf{0}$ ou $\mathbf{1}$, e resolve de maneira recorrente
3. nas folhas, aplica a operação booleana $\boldsymbol{op}$ diretamente

# Dependência conceitual

O algoritmo para a aplicação de operações booleanas entre OBDDs utiliza o conceito da Expansão de Shannon

# Definição: restrições

## Definição 6.9

Sejam $\phi$ uma expressão booleana e $p$ uma variável. Denotamos por $\phi[\mathbf{0}/p]$ a expressão booleana obtida substituindo-se todas as ocorrências de $p$ em $\phi$ por $\mathbf{0}$. A expressão $\phi[\mathbf{1}/p]$ é definida de maneira semelhante. As expressões $\phi[\mathbf{0}/p]$ e $\phi[\mathbf{1}/p]$ são chamadas de <u>restrições</u> em $\phi$ com relação à variável $p$.

# Exemplos de restrições

Para $\phi \equiv p \wedge (q \vee \neg p)$ tem-se:

- $\phi[0/p]$ é igual a $0 \wedge (q \vee \neg 0)$
  - que é semanticamente equivalente a $0$
- $\phi[1/p]$ é igual a $1 \wedge (q \vee \neg 1)$
  - que é semanticamente equivalente a $q$
- $\phi[0/q]$ é igual a $p \wedge (0 \vee \neg p)$
  - que é semanticamente equivalente a $\perp$
- $\phi[1/q]$ é igual a $p \wedge (1 \vee \neg p)$
  - que é semanticamente equivalente a $p$

# Uso das restrições

- As restrições permitem executar recorrências em expressões booleanas decompondo-as em expressões mais simples
- Se $p$ é uma variável em $\phi$, então $\phi$ é equivalente a $\neg p \wedge \phi[0/p] \vee p \wedge \phi[1/p]$
    - facilmente verificável
    - fazendo $p = 0$ resulta em $\phi[0/p]$
    - fazendo $p = 1$ resulta em $\phi[1/p]$

# Lema: Expansão de Shannon

## Lema 6.10

Para todas as expressões booleanas $\phi$ e todas as variáveis $p$ (mesmo as que não ocorrem em $\phi$), tem-se a chamada Expansão de Shannon:

$$\phi \equiv \neg p \wedge \phi[0/p] \vee p \wedge \phi[1/p]$$

# Uso no algoritmo de aplicação

A Expansão de Shannon permite expressar qualquer operador da gramática da Lógica Proposicional:

$$\boldsymbol{\phi \; op \; \psi \equiv \neg p_i \land (\phi[0/p_i] \; op \; \psi[0/p_i]) \lor p_i \land (\phi[1/p_i] \; op \; \psi[1/p_i])}$$

E, no algoritmo de aplicação, é usada para eliminar as variáveis com a aplicação das restrições

# Programação dinâmica

- O algoritmo de aplicação também utiliza programação dinâmica para melhorar a eficiência
  - recursão com tabela para armazenar valores já calculados

No pseudocódigo do algoritmo, a seguir, essa tabela é referenciada como $C$ (de *cache*)

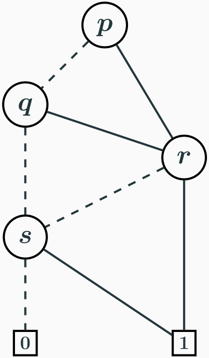
# Pseudocódigo de APPLY

**precondição:** recebe um operador lógico e os nós raízes de dois diagramas com ordens compatíveis
**pós-condição:** devolve o identificador do nó canônico do resultado da operação

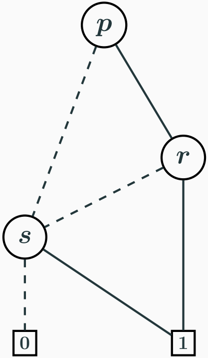
**função** APPLY$(op, n_\phi, n_\psi)$
  $v_\phi \leftarrow$ VAR$(n_\phi)$
  $v_\psi \leftarrow$ VAR$(n_\psi)$
  **se** $(v_\phi \in \{0,1\}) \land (v_\psi \in \{0,1\})$ **então** ▷ se ambos são nós terminais
    $r \leftarrow n_\phi \ op \ n_\psi$ ▷ aplica a operação diretamente
    **devolva** GETNODE$(r,$ NULL, NULL$)$
  $r \leftarrow C(op, n_\phi, n_\psi)$ ▷ verificação da programação dinâmica
  **se** r = NULL **então**
    **se** $v_\phi = v_\psi$ **então** ▷ se têm a mesma variável
      $i_l \leftarrow$ APPLY$(op,$ LO$(n_\phi),$ LO$(n_\psi))$
      $i_h \leftarrow$ APPLY$(op,$ HI$(n_\phi),$ HI$(n_\psi))$
      $r \leftarrow$ GETNODE$(v_\phi, i_l, i_h)$
    **else**
      **se** $v_\phi \prec v_\psi$ **então** ▷ se $v_\phi$ ocorre antes de $v_\psi$
        $i_l \leftarrow$ APPLY$(op,$ LO$(n_\phi), n_\psi)$
        $i_h \leftarrow$ APPLY$(op,$ HI$(n_\phi), n_\psi)$
        $r \leftarrow$ GETNODE$(v_\phi, i_l, i_h)$
      **else** ▷ se $v_\phi$ ocorre após $v_\psi$
        $i_l \leftarrow$ APPLY$(op, n_\phi,$ LO$(n_\psi))$
        $i_h \leftarrow$ APPLY$(op, n_\phi,$ HI$(n_\psi))$
        $r \leftarrow$ GETNODE$(v_\psi, i_l, i_h)$
    $C \leftarrow C \cup \{(\langle op, n_\phi, n_\psi \rangle, r)\}$ ▷ atualização da tabela da progamação dinâmica
  **devolva** $r$

# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



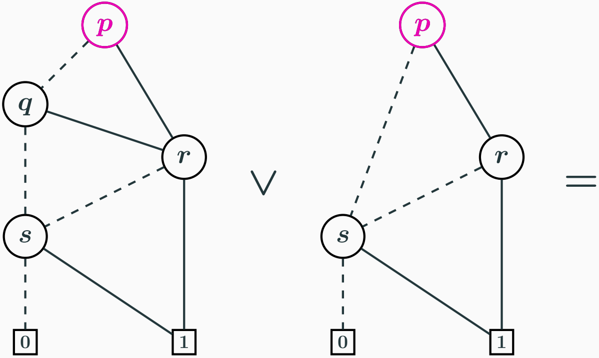

$\vee$




$=$

# Ilustração do algoritmo de aplicação

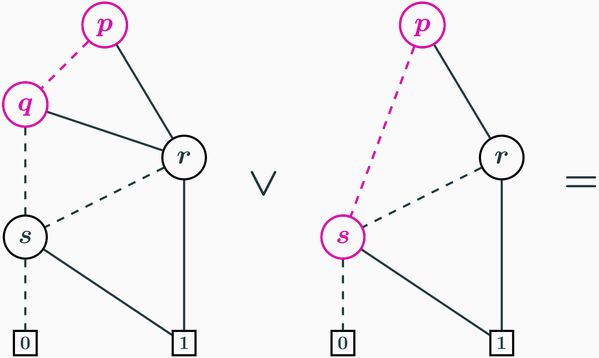
Ordem: $[p, q, r, s]$




mesma variável: recorrência em ambos diagramas

# Ilustração do algoritmo de aplicação

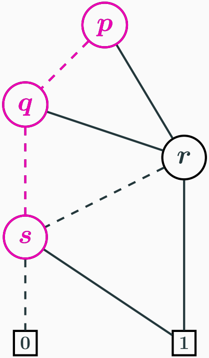
Ordem: $[p, q, r, s]$



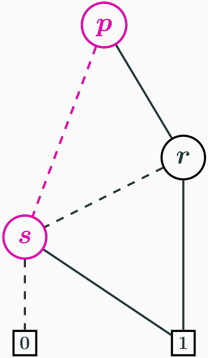

$q \prec s$: recorrência no diagrama da esquerda

# Ilustração do algoritmo de aplicação

Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$




$\vee$

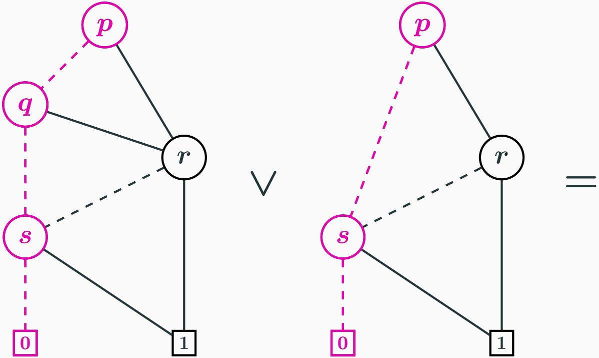



$=$

mesma variável: recorrência em ambos diagramas

# Ilustração do algoritmo de aplicação

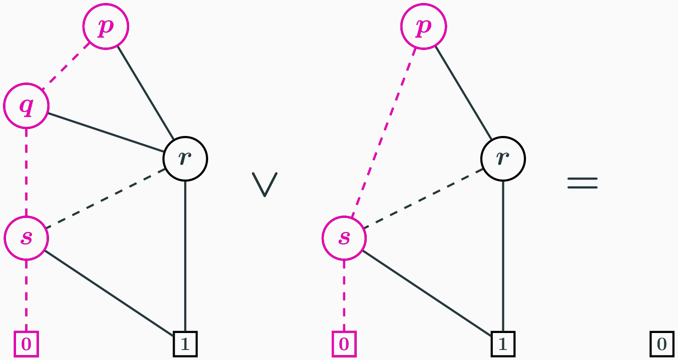
Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$




nós terminais: aplica operação $(\mathbf{0} \vee \mathbf{0} = \mathbf{0})$

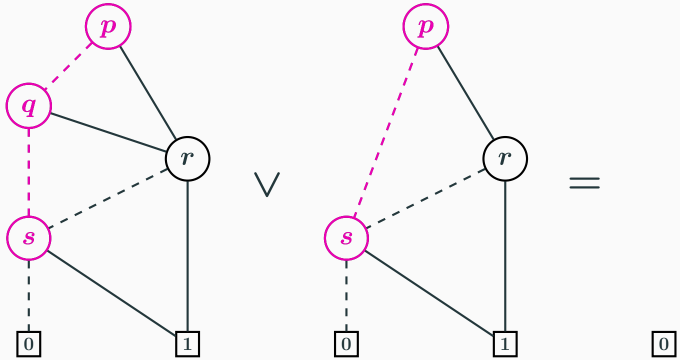
# Ilustração do algoritmo de aplicação

Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$



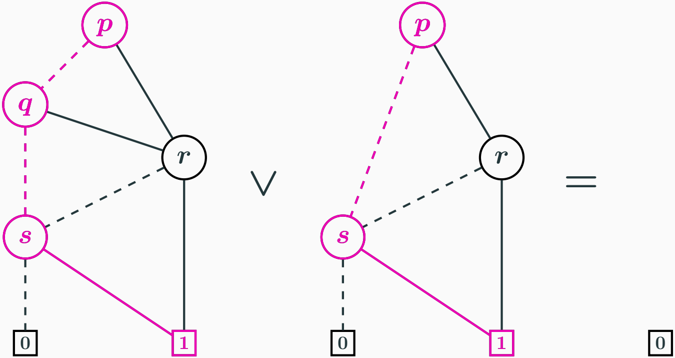
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



# Ilustração do algoritmo de aplicação

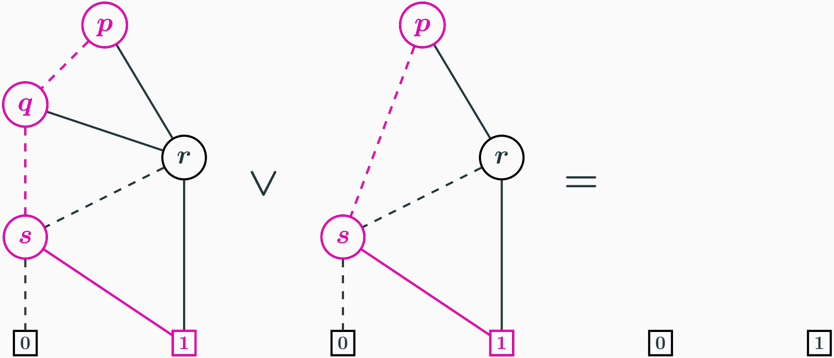
Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$




nós terminais: aplica operação $(\mathbf{1} \vee \mathbf{1} = \mathbf{1})$

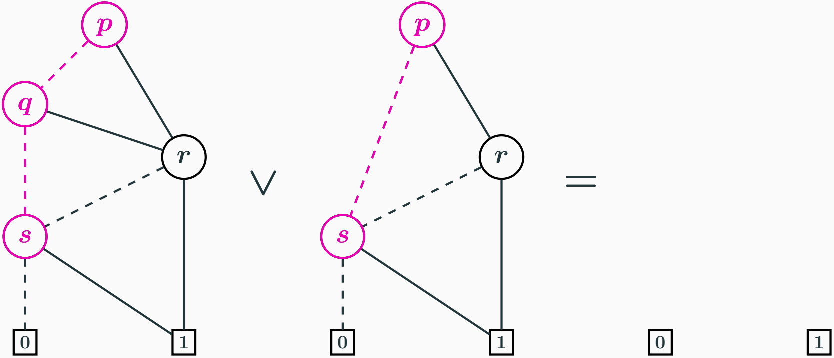
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



# Ilustração do algoritmo de aplicação

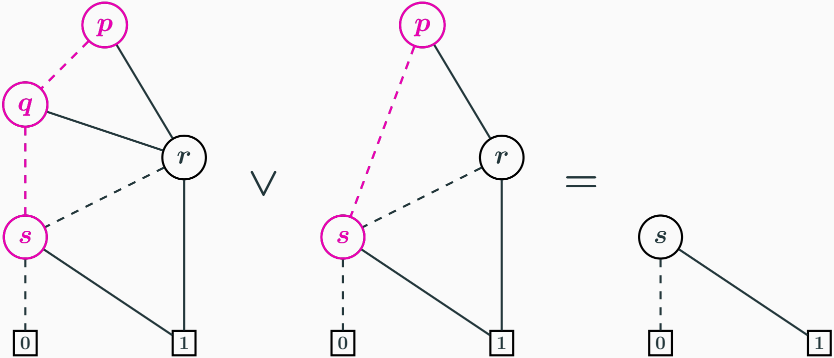
Ordem: $[p, q, r, s]$




Conclusão parcial: cria/compartilha subdiagrama

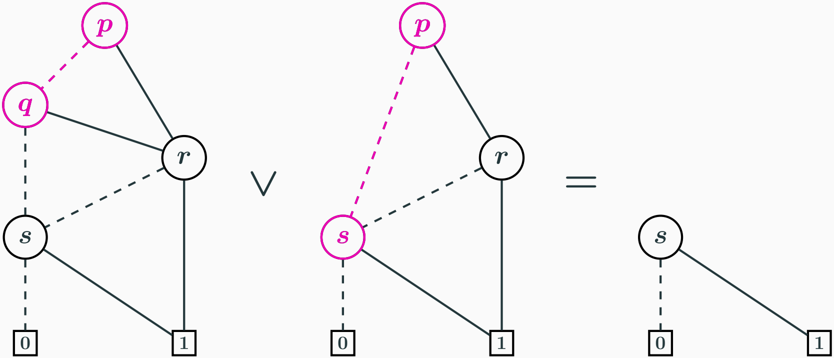
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



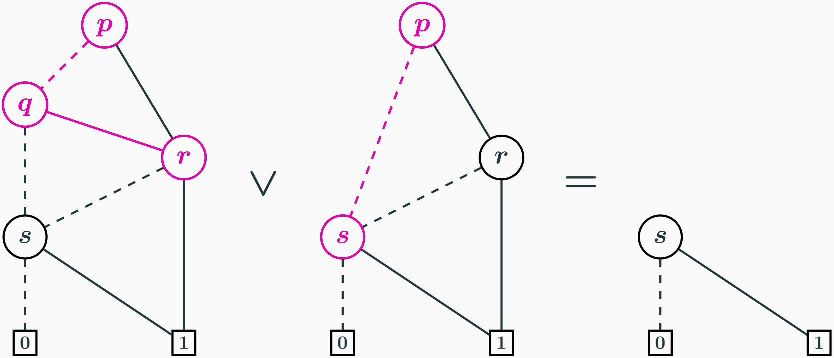
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



# Ilustração do algoritmo de aplicação

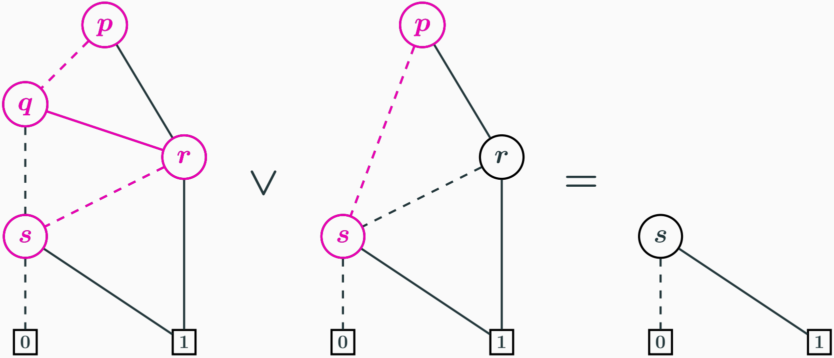
Ordem: $[p, q, r, s]$




$r \prec s$: recorrência no diagrama da esquerda

# Ilustração do algoritmo de aplicação

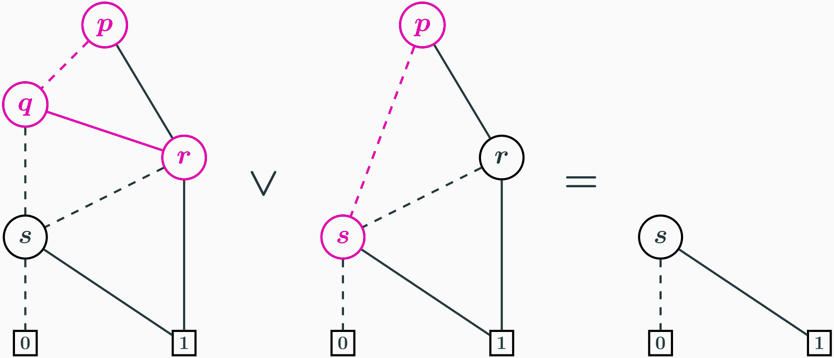
Ordem: $[p, q, r, s]$




programação dinâmica: operação entre subdiagramas já calculada

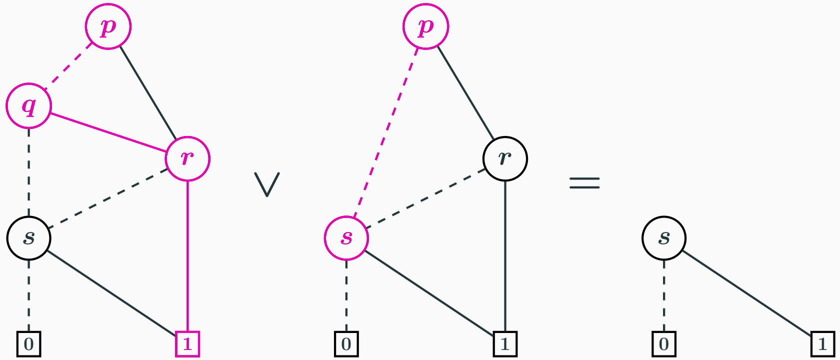
# Ilustração do algoritmo de aplicação

Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$



# Ilustração do algoritmo de aplicação

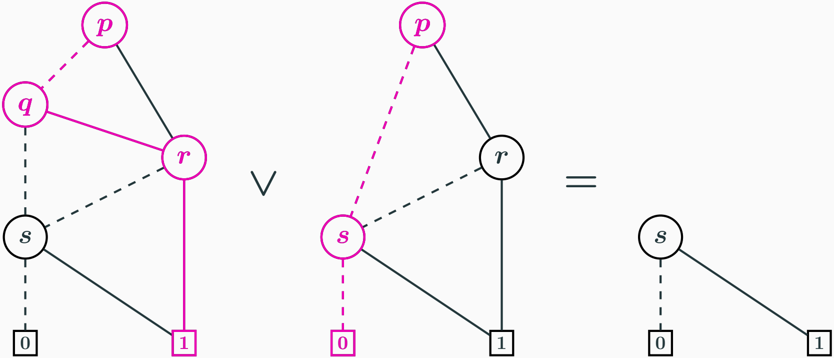
Ordem: $[p, q, r, s]$




$\mathbf{1} \succ s$: recorrência no diagrama da direita

# Ilustração do algoritmo de aplicação

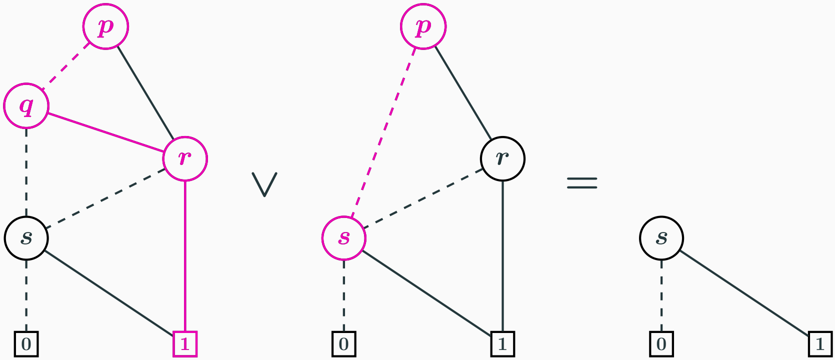
Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$




nós terminais: aplica operação $(\mathbf{1} \vee \mathbf{0} = \mathbf{1})$

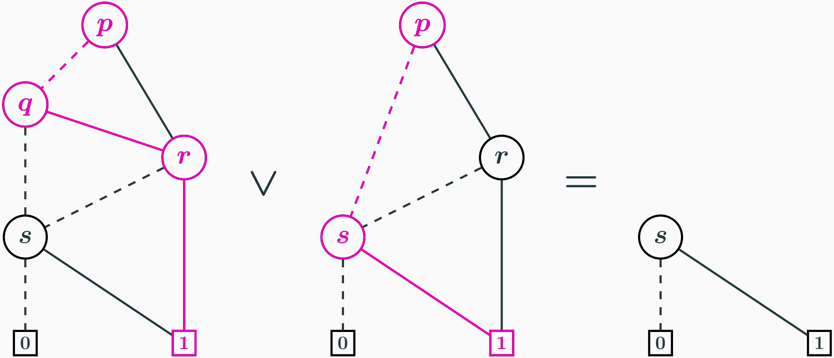
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



# Ilustração do algoritmo de aplicação

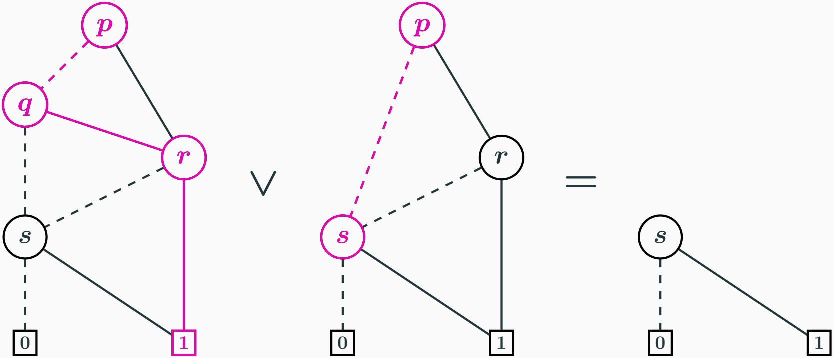
Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$




nós terminais: aplica operação $(\mathbf{1} \vee \mathbf{1} = \mathbf{1})$

# Ilustração do algoritmo de aplicação

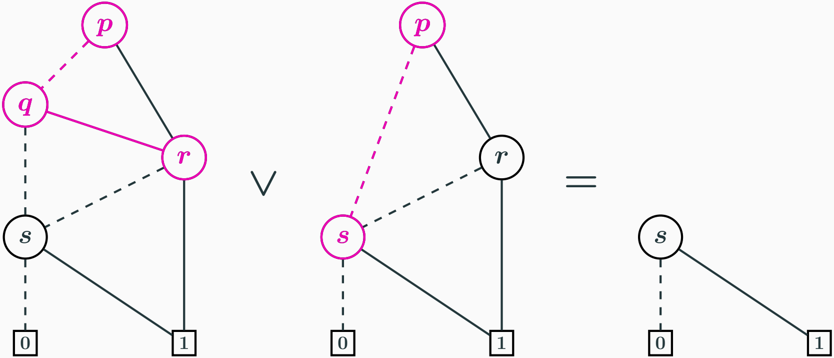
Ordem: $[p, q, r, s]$




simplificação: nó $s$ que sempre chega a $1$ ignorado

# Ilustração do algoritmo de aplicação

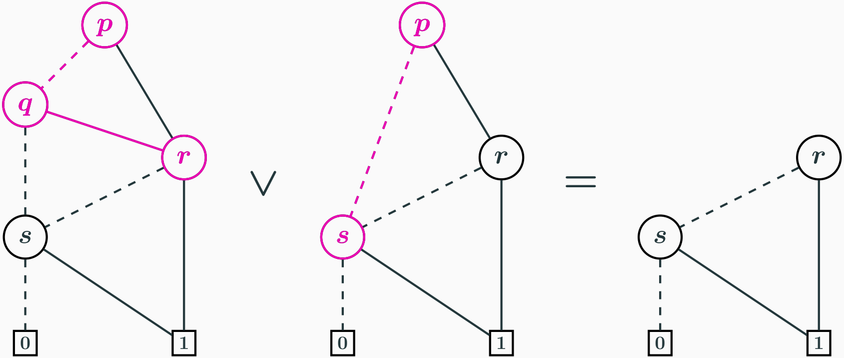
Ordem: $[p, q, r, s]$




Conclusão parcial: cria/compartilha subdiagrama

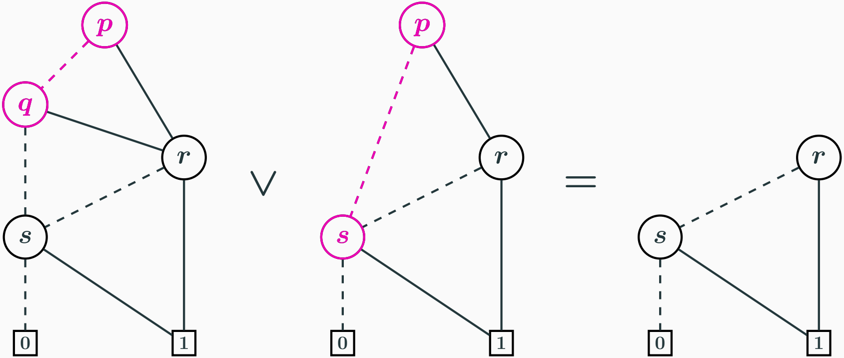
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



# Ilustração do algoritmo de aplicação

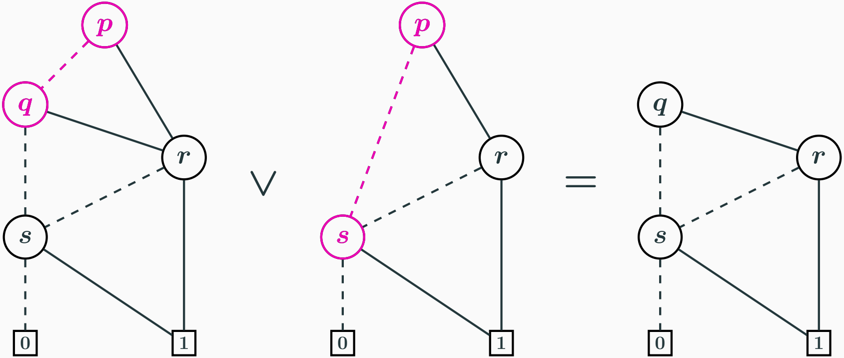
Ordem: $[p, q, r, s]$




Conclusão parcial: cria/compartilha subdiagrama

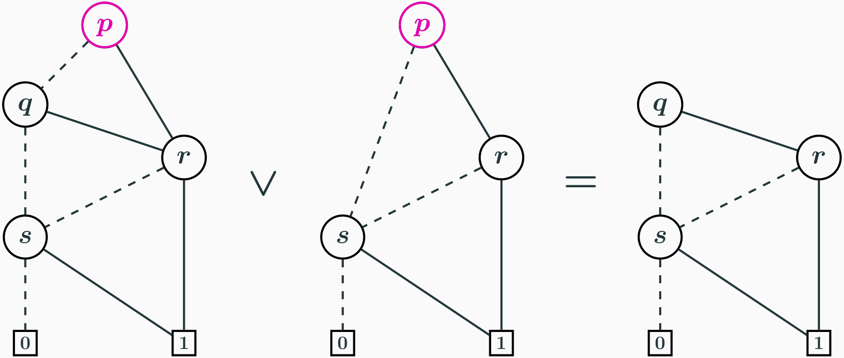
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



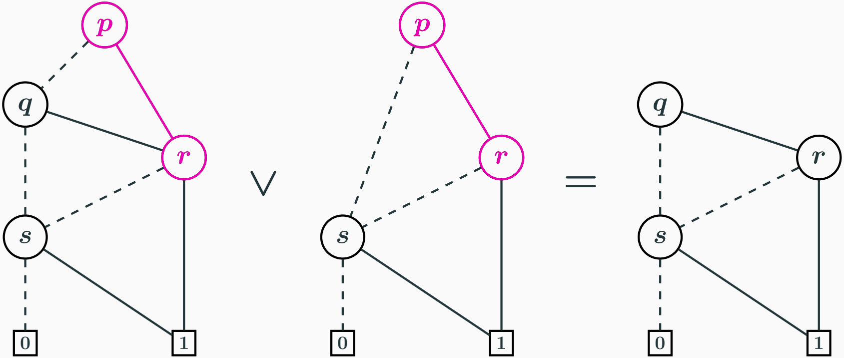
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



# Ilustração do algoritmo de aplicação

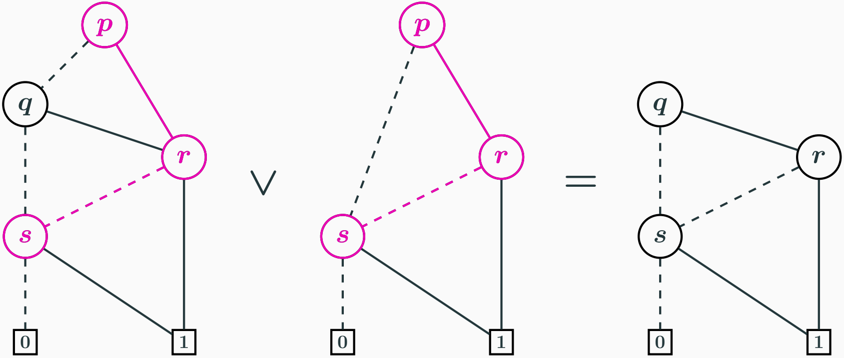
Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$




mesma variável: recorrência em ambos diagramas

# Ilustração do algoritmo de aplicação

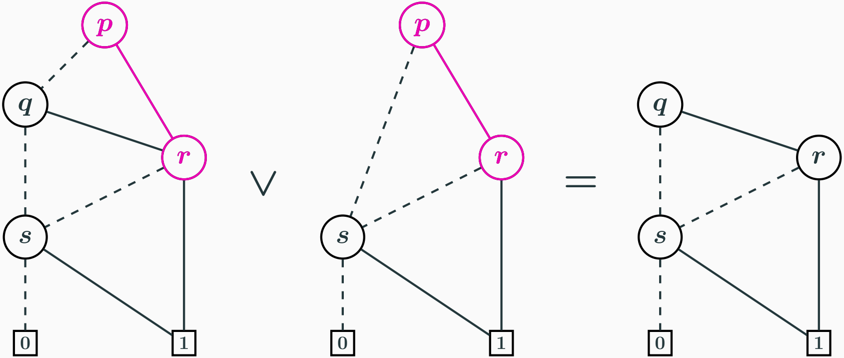
Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$




programação dinâmica: operação entre subdiagramas já calculada

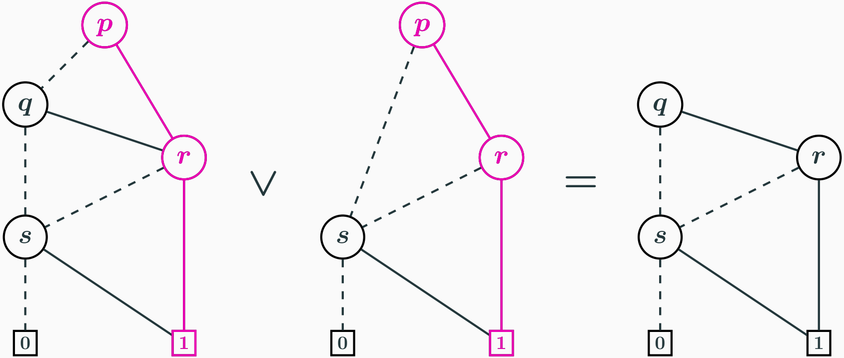
# Ilustração do algoritmo de aplicação

Ordem: $[p, q, r, s]$



# Ilustração do algoritmo de aplicação

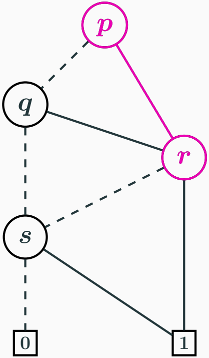
Ordem: $[p, q, r, s]$



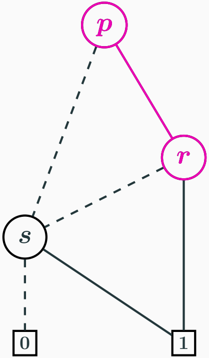

programação dinâmica: operação entre subdiagramas já calculada

# Ilustração do algoritmo de aplicação

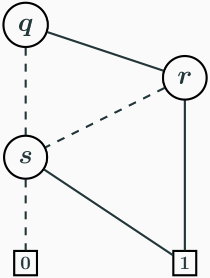
Ordem: $[p, q, r, s]$




$\vee$

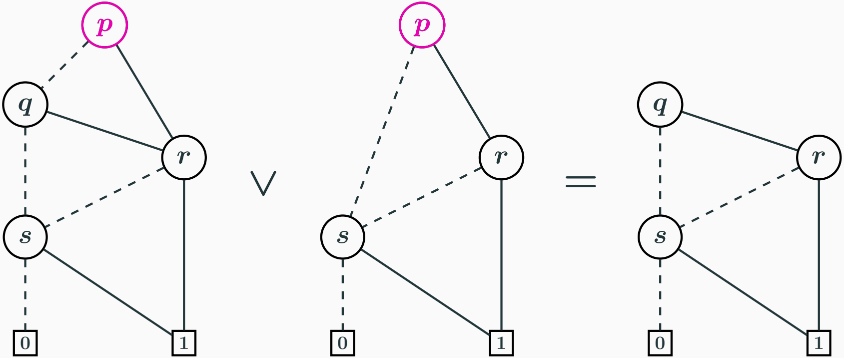



$=$



# Ilustração do algoritmo de aplicação

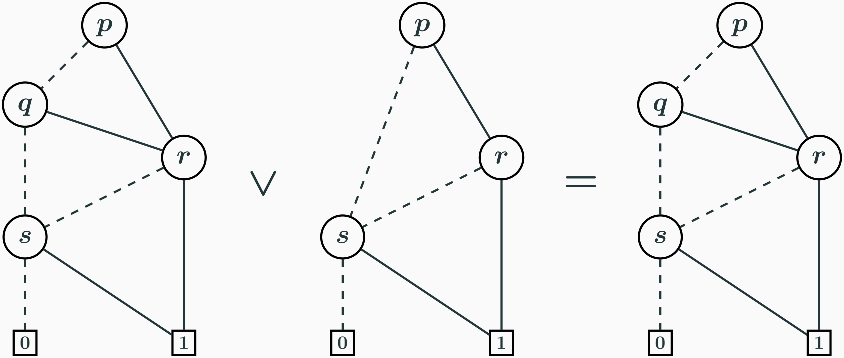
Ordem: $[p, q, r, s]$




Conclusão parcial: cria/compartilha subdiagrama

# Ilustração do algoritmo de aplicação

Ordem: $[\boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}]$



# Outros algoritmos para ROBDDs

Há também esses outros dois algoritmos importantes:

- **Restrição**. Permite eliminar variáveis em diagramas
- **Existência**. Permite utilizar quantificadores em expressões

# O algoritmo de restrição

O algoritmo de restrição (RESTRICT) calcula o RODDB que representem $\phi[\mathbf{0}/p]$ ou $\phi[\mathbf{1}/p]$. É bem simples, e funciona assim:

- Para cada nó $n$ marcado com a variável $p$, as arestas que entram são redirecionadas
  - para $\text{LO}(n)$ se o valor de restrição é $\mathbf{0}$
  - para $\text{HI}(n)$ se o valor de restrição é $\mathbf{1}$
- Então, o algoritmo de redução é chamado para reduzir o OBDD resultante

# O algoritmo de existência

O algoritmo de existência (EXISTS) representa expressões em termos de subconjuntos de restrições. Funciona assim:

- Seja uma expressão $\boldsymbol{p} \vee (\neg \boldsymbol{q} \wedge \boldsymbol{r})$. Ela só é verdadeira se $\boldsymbol{p} = \mathbf{1}$ ou se $\boldsymbol{y} = \mathbf{0}$ e $\boldsymbol{r} = \mathbf{1}$
    - ou seja, tratam-se de restrições sobre $\boldsymbol{p}$, $\boldsymbol{q}$ e $\boldsymbol{r}$
- Pode-se expressar o relaxamento em um subconjunto de variáveis
    - escrevendo $\exists \boldsymbol{p}.\boldsymbol{\phi}$ para a função booleana $\boldsymbol{\phi}$ com restrição sobre $\boldsymbol{p}$ relaxada
    - formalmente $\exists \boldsymbol{p}.\boldsymbol{\phi} \stackrel{\text{def}}{=} \boldsymbol{\phi}[\mathbf{0}/\boldsymbol{p}] \vee \boldsymbol{\phi}[\mathbf{1}/\boldsymbol{p}]$

# O que significa

Essencialmente, $\exists \boldsymbol{p}.\boldsymbol{\phi}$ significa que $\boldsymbol{\phi}$ é verdadeira se puder ser feita verdadeira para $\boldsymbol{p = 0}$ ou para $\boldsymbol{p = 1}$

# Analogamente

Analogamente, $\forall \boldsymbol{p}.\boldsymbol{\phi}$ significa que $\boldsymbol{\phi}$ é verdadeira se puder ser feita verdadeira tanto para $\boldsymbol{p} = \mathbf{0}$ como para $\boldsymbol{p} = \mathbf{1}$

E por isso o quantificador booleano $\forall$ é o dual de $\exists$:

$$\forall \boldsymbol{p}.\boldsymbol{\phi} \stackrel{\text{def}}{=} \boldsymbol{\phi}[\mathbf{0}/\boldsymbol{p}] \wedge \boldsymbol{\phi}[\mathbf{1}/\boldsymbol{p}]$$

# O algoritmo EXISTS

Assim, o algoritmo EXISTS pode ser implementado em termos dos algoritmos APPLY e RESTRICT da seguinte forma:

$$\textsc{apply}(\vee, \textsc{restrict}(\mathbf{0}, \boldsymbol{p}, \boldsymbol{B_\phi}), \textsc{restrict}(\mathbf{1}, \boldsymbol{p}, \boldsymbol{B_\phi}))$$

# Construção de ROBDDs

Formas de construção de ROBDDs com os algoritmos

| Fórmula booleana $\phi$ | ROBDD $B_\phi representante$ |
|---|---|
| $\mathbf{0}$ | $B_\mathbf{0}$ |
| $\mathbf{1}$ | $B_\mathbf{1}$ |
| $p$ | $B_p$ |
| $\neg\phi$ | trocar $\mathbf{0}$ por $\mathbf{1}$ e vice-versa em $B_\phi$ |
| $\phi \vee \psi$ | $\text{APPLY}(\vee, B_\phi, B_\psi)$ |
| $\phi \wedge \psi$ | $\text{APPLY}(\wedge, B_\phi, B_\psi)$ |
| $\phi \oplus \psi$ | $\text{APPLY}(\oplus, B_\phi, B_\psi)$ |
| $\phi[\mathbf{1}/p]$ | $\text{RESTRICT}(\mathbf{1}, p, B_\phi)$ |
| $\phi[\mathbf{0}/p]$ | $\text{RESTRICT}(\mathbf{0}, p, B_\phi)$ |
| $\exists p.\phi$ | $\text{APPLY}(\vee, B_{\phi[\mathbf{0}/p]}, B_{\phi[\mathbf{1}/p]})$ |
| $\forall p.\phi$ | $\text{APPLY}(\wedge, B_{\phi[\mathbf{0}/p]}, B_{\phi[\mathbf{1}/p]})$ |

# Desempenho dos algoritmos

Formas de construção de ROBDDs com os algoritmos

| Algoritmo | Complexidade no Tempo |
|---|---|
| REDUCE | $\boldsymbol{O(\lvert B\rvert \times \log \lvert B\rvert)}$ |
| APPLY | $\boldsymbol{O(\lvert B_\phi\rvert \times \lvert B_\psi\rvert)}$ |
| RESTRICT | $\boldsymbol{O(\lvert B\rvert \times \log \lvert B\rvert)}$ |
| EXISTS | NP completo |

# Biblioteca de ROBDDs para o EP

- BDDs from Python EDA: `http://pyeda.readthedocs.org/en/latest/bdd.html`
  - Python EDA é uma biblioteca Python para projetos de automação
  - usando álgebra booleana

# Uso é simples

```
>>> from pyeda.inter import *

>>> f = expr("a & b | a & c | b & c")
>>> f
Or(And(a, b), And(a, c), And(b, c))
>>> f = expr2bdd(f)
>>> f
<pyeda.boolalg.bdd.BinaryDecisionDiagram at 0x7f556874e
```

# Verificações

```
>>> f = ~a & ~b | ~a & b | a & ~b | a & b
>>> f
1
>>> f.is_one()
True
>>> g = (~a | ~b) & (~a | b) & (a | ~b) & (a | b)
>>> g
0
>>> g.is_zero()
True
```

# Operações

```
>>> f = expr("a & b | a & c | b & c")

>>> f.restrict({a: 0})
<pyeda.boolalg.bdd.BinaryDecisionDiagram at 0x7f556874e
>>> f.restrict({a: 1, b: 0})
c
>>> f.restrict({a: 1, b: 1})
1
```