Conceitos Avançados de Prolog

- Esta aula trata de conceitos avançados em Prolog, tais como
  - *Backtracking*
  - Corte
  - Negação por falha
  - Entrada e saída
  - Predicados Adicionais

Inteligência Artificial

José Augusto Baranauskas
Departamento de Física e Matemática – FFCLRP-USP
Sala 228 – Bloco P2 – Fone (16) 602-4439

E-mail: augusto@ffclrp.usp.br
URL: http://www.fmrp.usp.br/augusto

Backtracking

- Normalmente Prolog efetua um retrocesso (*backtrack*) sempre que necessário na tentativa de satisfazer uma cláusula com busca em profundidade

- Por um lado, o ***backtracking* automático** é um conceito muito útil, já que ele libera o programador de implementá-lo explicitamente

- Por outro lado, o uso indiscriminado de *backtracking* pode causar ineficiência em um programa

Backtracking

```
gosta(maria,comida).
gosta(maria,vinho).
gosta(joao,vinho).
gosta(joao,maria).
```

```
?- gosta(maria,X), gosta(joao,X).
```

Backtracking

gosta(maria,comida).←
gosta(maria,vinho).
gosta(joao,vinho).
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

1. A primeira cláusula tem sucesso, e X é instanciado com “comida”

Backtracking

gosta(maria,comida).
gosta(maria,vinho).
gosta(joao,vinho).
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

comida comida

1. A primeira cláusula tem sucesso, e X é instanciado com "comida"
2. A seguir, Prolog tenta provar a segunda cláusula...

Backtracking

gosta(maria,comida).←
gosta(maria,vinho). ←×
gosta(joao,vinho).
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

1. A primeira cláusula tem sucesso, e X é instanciado com "comida"
2. A seguir, Prolog tenta provar a segunda cláusula...

Backtracking

gosta(maria,comida).←
gosta(maria,vinho).
gosta(joao,vinho). ←×
gosta(joao,maria).

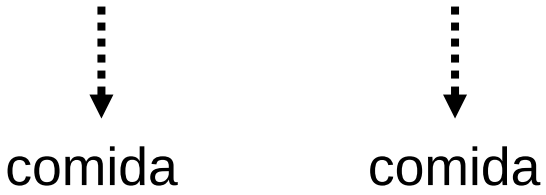
?- gosta(maria,X), gosta(joao,X).

1. A primeira cláusula tem sucesso, e X é instanciado com "comida"
2. A seguir, Prolog tenta provar a segunda cláusula...

Backtracking

gosta(maria,comida).←
gosta(maria,vinho).
gosta(joao,vinho).
gosta(joao,maria). ←×

?- gosta(maria,X), gosta(joao,X).



1. A primeira cláusula tem sucesso, e X é instanciado com "comida"
2. A seguir, Prolog tenta provar a segunda cláusula
3. A segunda cláusula falha

Backtracking

gosta(maria,comida).←
gosta(maria,vinho).
gosta(joao,vinho).
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

1. A primeira cláusula tem sucesso, e X é instanciado com "comida"
2. A seguir, Prolog tenta provar a segunda cláusula
3. A segunda cláusula falha
4. Ocorre *backtrack*, Prolog desinstancia X e tenta satisfazer novamente a primeira cláusula

Backtracking

gosta(maria,comida).
gosta(maria,vinho). ←
gosta(joao,vinho).
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

5. A primeira cláusula tem sucesso novamente e X é instanciado com “vinho”

Backtracking

gosta(maria,comida). ←×
gosta(maria,vinho). ←
gosta(joao,vinho).
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

5. A primeira cláusula tem sucesso novamente e X é instanciado com “vinho”
6. A seguir, Prolog tenta provar a segunda cláusula...

Backtracking

gosta(maria,comida).
gosta(maria,vinho). ← ←×
gosta(joao,vinho).
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

5. A primeira cláusula tem sucesso novamente e X é instanciado com “vinho”
6. A seguir, Prolog tenta provar a segunda cláusula...

Backtracking

gosta(maria,comida).
gosta(maria,vinho). ←
gosta(joao,vinho). ←
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

5. A primeira cláusula tem sucesso novamente e X é instanciado com “vinho”
6. A seguir, Prolog tenta provar a segunda cláusula
7. A segunda cláusula tem sucesso
8. Prolog notifica o usuário e aguarda

X = vinho

Backtracking

gosta(maria,comida).
gosta(maria,vinho). ←
gosta(joao,vinho). ←
gosta(joao,maria).

?- gosta(maria,X), gosta(joao,X).

5. A primeira cláusula tem sucesso novamente e X é instanciado com “vinho”
6. A seguir, Prolog tenta provar a segunda cláusula
7. A segunda cláusula tem sucesso
8. Prolog notifica o usuário e aguarda

X = vinho ;

9. Usuário deseja outra solução: fica como exercício

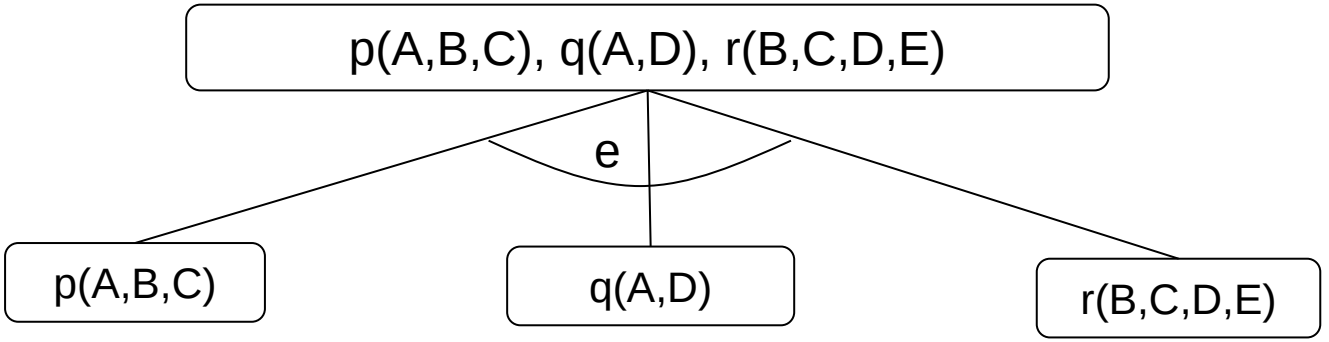
Importante

- Sempre que uma cláusula falha (consequentemente ocorre um *backtrack*), as variáveis que se tornaram instanciadas naquela cláusula tornam-se variáveis livres novamente

- As variáveis instanciadas em cláusulas anteriores permanecem instanciadas

- Quando uma solução é encontrada e o usuário solicita outra solução (digitando **;**), as variáveis instanciadas na cláusula atual tornam-se variáveis livres e o processo de prova reinicia a partir da situação corrente, como ilustrado a seguir

Backtracking (representação em grafo E/OU)



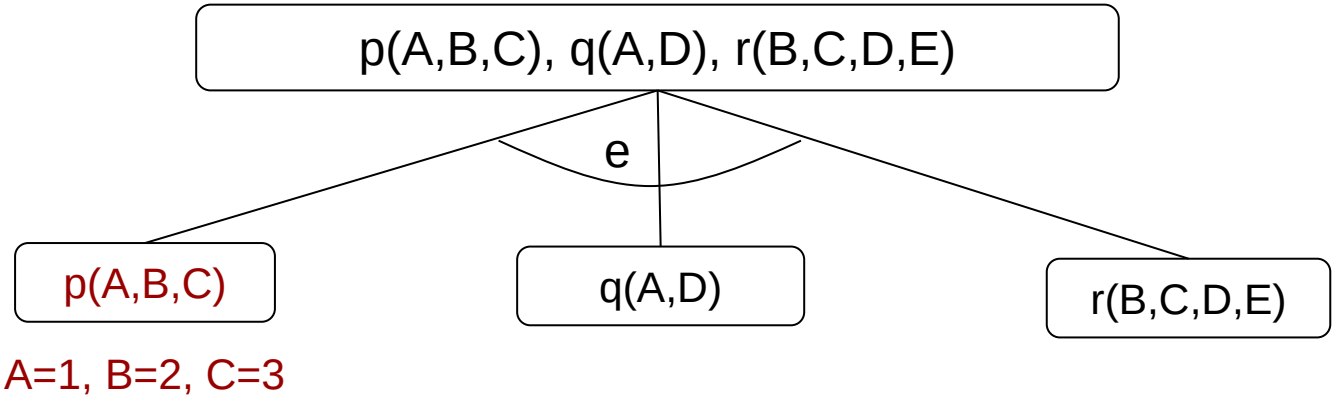
p(1,2,3).
p(3,5,7).
p(2,4,6).

q(5,10).
q(1,20).
q(3,7).

r(1,2,3,4).
r(5,7,7,9).
r(2,4,6,8).

Variáveis Livres: A, B, C, D, E

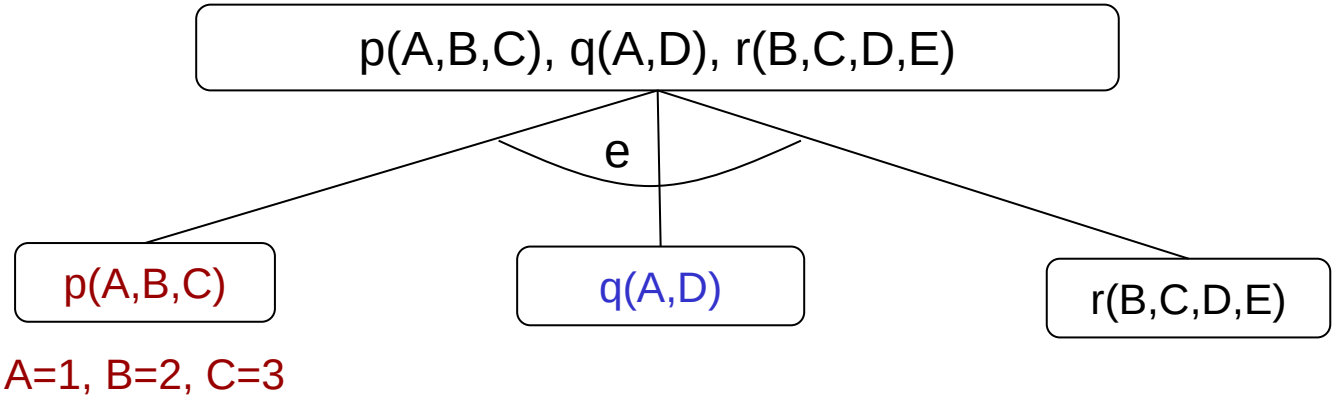
Backtracking



→	p(1,2,3).	q(5,10).	r(1,2,3,4).
	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

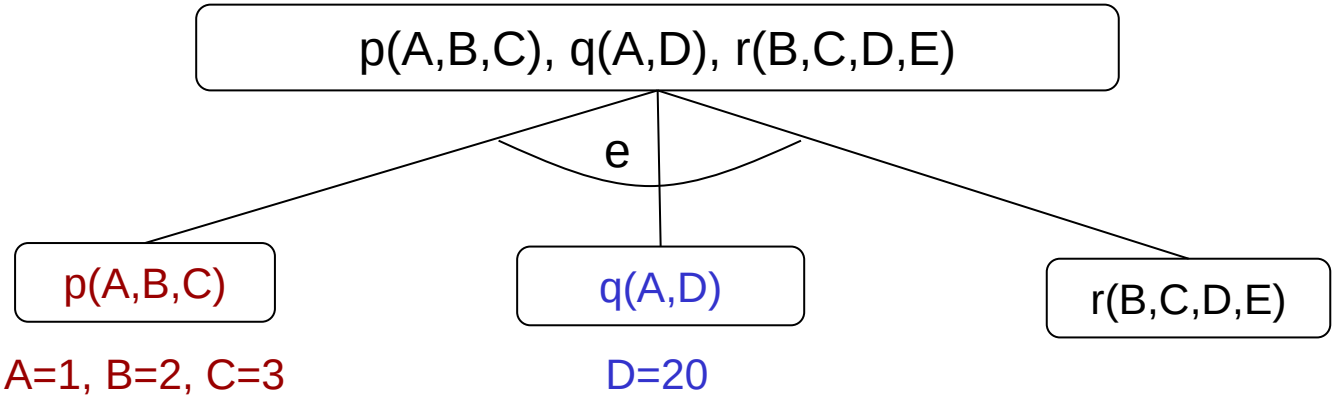
Backtracking



→	p(1,2,3).	×→ q(5,10).	r(1,2,3,4).
	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

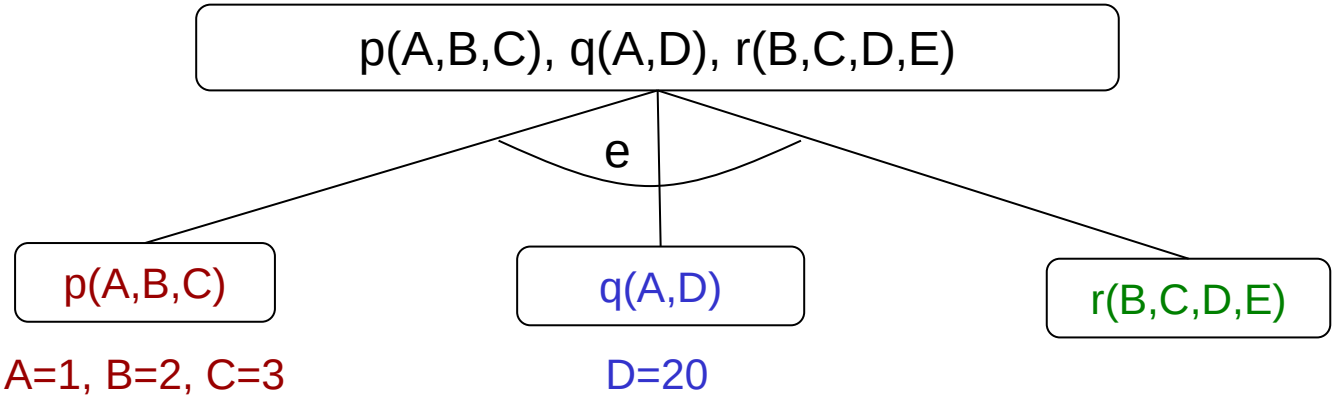
Backtracking



→ p(1,2,3).	q(5,10).	r(1,2,3,4).
p(3,5,7).	→ q(1,20).	r(5,7,7,9).
p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: E

Backtracking



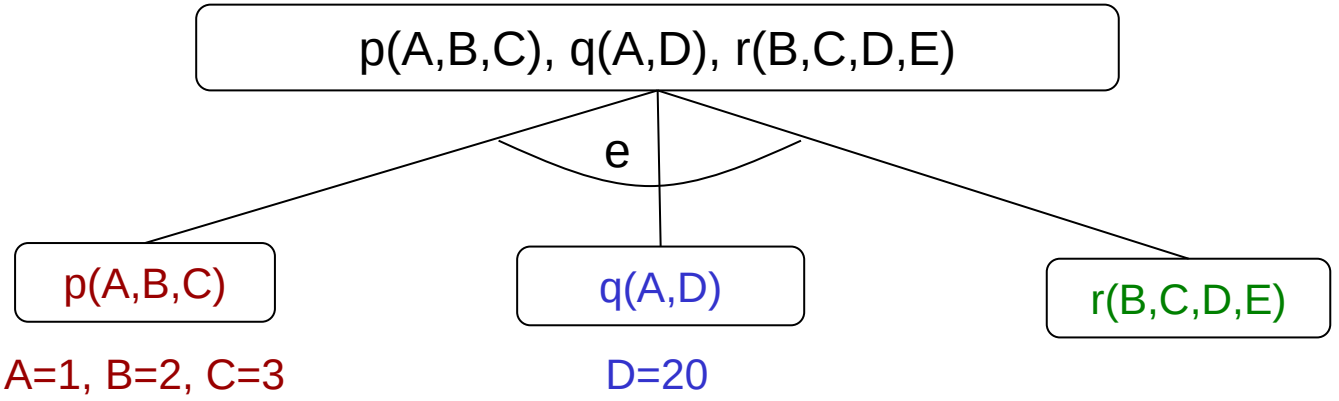
→ p(1,2,3).
p(3,5,7).
p(2,4,6).

q(5,10).
→ q(1,20).
q(3,7).

× → r(1,2,3,4).
r(5,7,7,9).
r(2,4,6,8).

Variáveis Livres: E

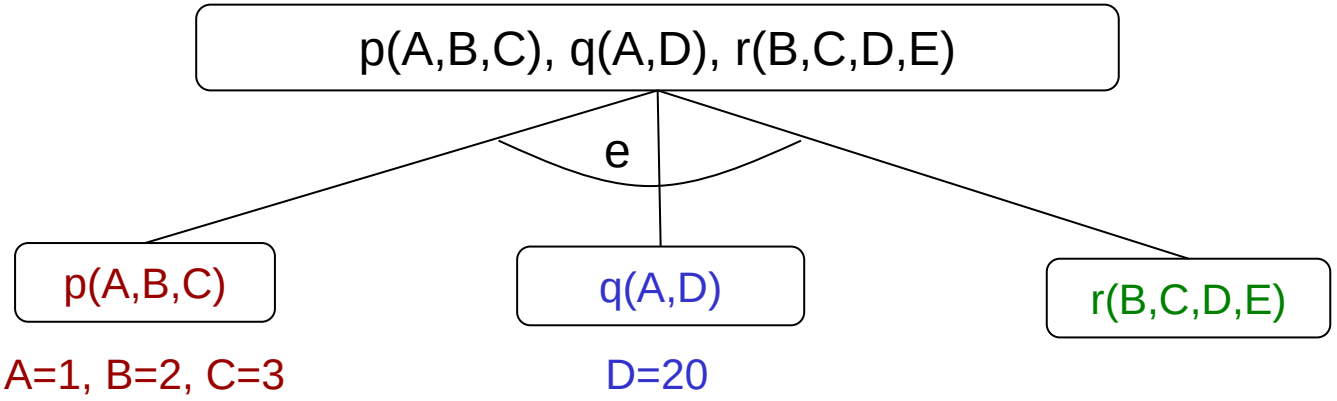
Backtracking



→ p(1,2,3).	q(5,10).	r(1,2,3,4).
p(3,5,7).	→ q(1,20).	×→ r(5,7,7,9).
p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: E

Backtracking

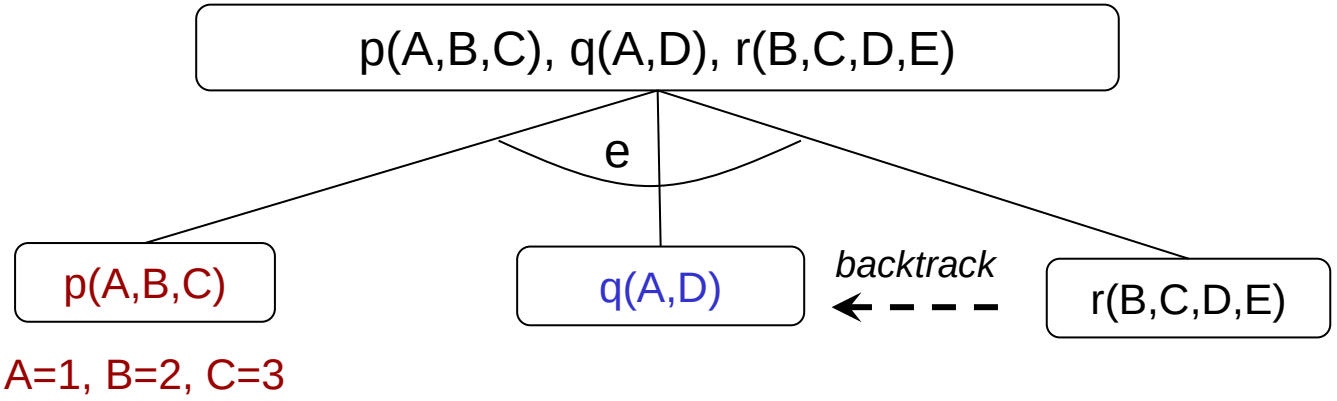


→ p(1,2,3).	q(5,10).	r(1,2,3,4).
p(3,5,7).	→ q(1,20).	r(5,7,7,9).
p(2,4,6).	q(3,7).	×→ r(2,4,6,8).

Variáveis Livres: E

Como não há outros modos de provar a relação **r/4**, o único modo de encontrar uma solução consiste em retroceder (*backtracking*) para a cláusula anterior

Backtracking

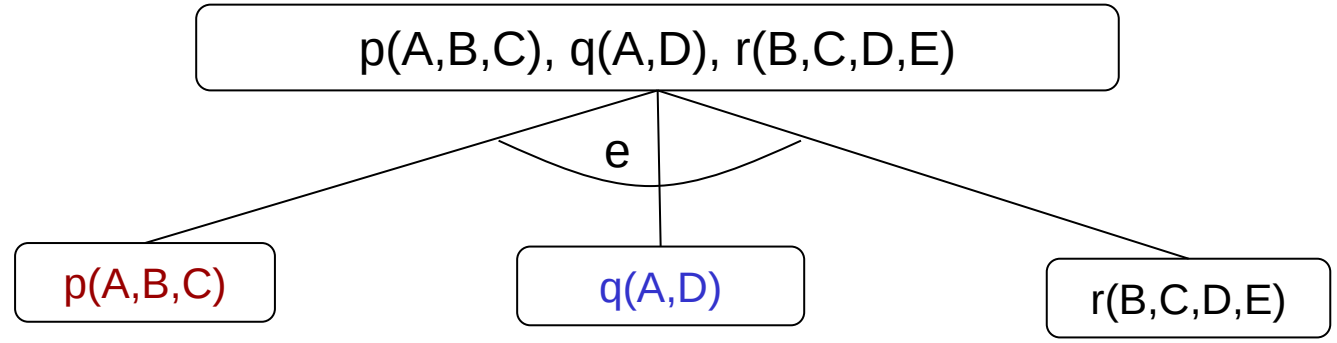


→ p(1,2,3).	q(5,10).	r(1,2,3,4).
p(3,5,7).	→ q(1,20).	r(5,7,7,9).
p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

No *backtracking*, as variáveis instanciadas na cláusula q/2 tornam-se variáveis livres e Prolog tenta encontrar uma outra solução para q(1,D)

Backtracking



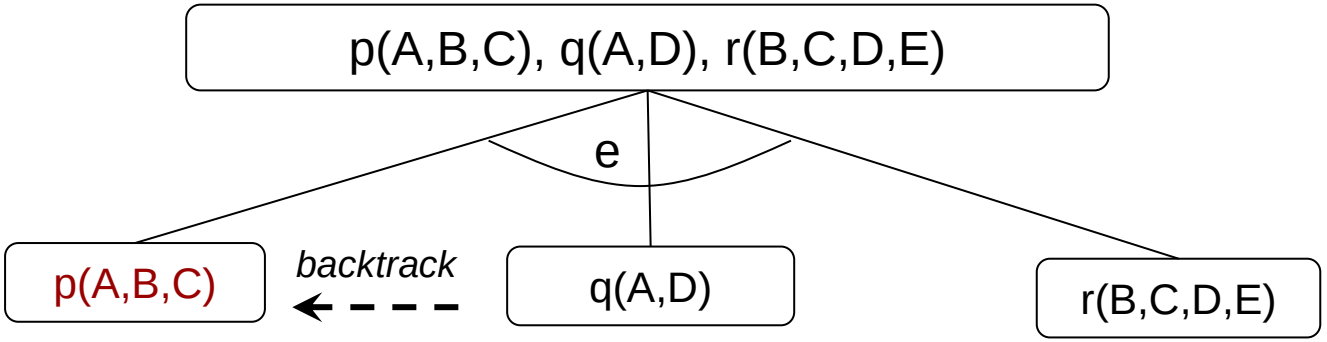
A=1, B=2, C=3

→ p(1,2,3).	q(5,10).	r(1,2,3,4).
p(3,5,7).	q(1,20).	r(5,7,7,9).
p(2,4,6).	×→ q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

Como não há outros modos de provar a relação **q/2**, o único modo de encontrar uma solução consiste em retroceder (*backtracking*) para a cláusula anterior

Backtracking

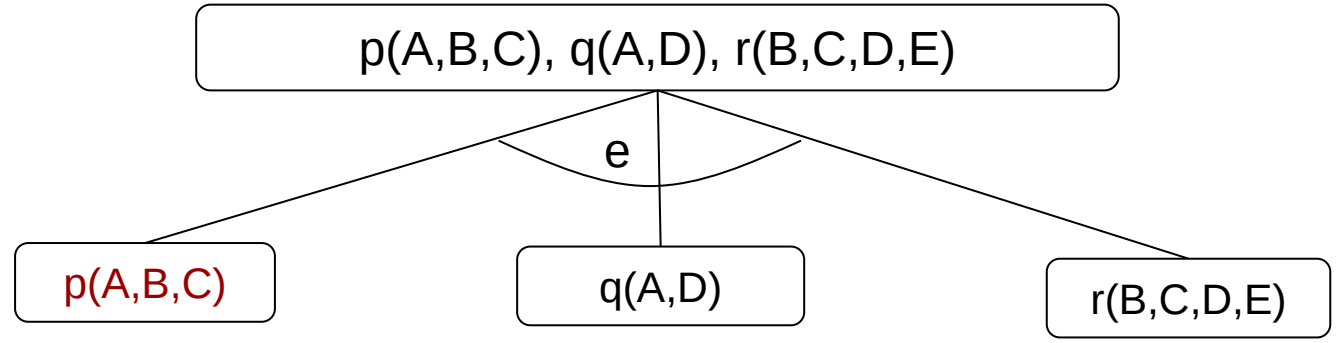


→	p(1,2,3).	q(5,10).	r(1,2,3,4).
	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: A, B, C, D, E

No *backtracking*, as variáveis instanciadas na cláusula p/3 tornam-se variáveis livres e Prolog tenta encontrar uma outra solução para p(A,B,C)

Backtracking

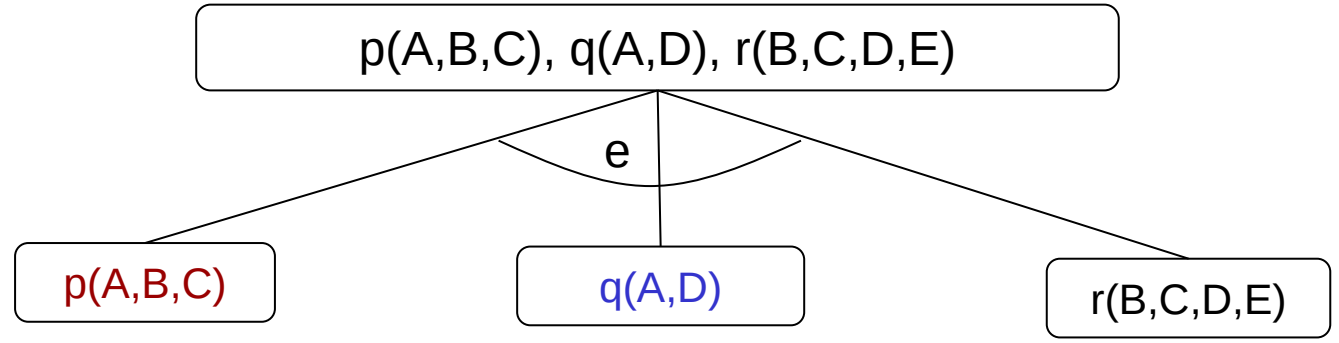


A=3, B=5, C=7

	p(1,2,3).	q(5,10).	r(1,2,3,4).
→	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

Backtracking

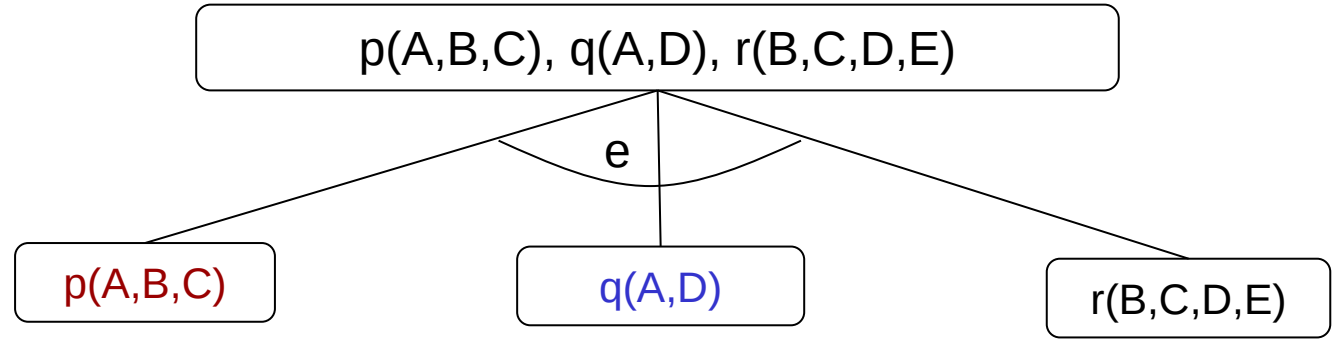


A=3, B=5, C=7

	p(1,2,3).	✗→ q(5,10).	r(1,2,3,4).
→	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

Backtracking

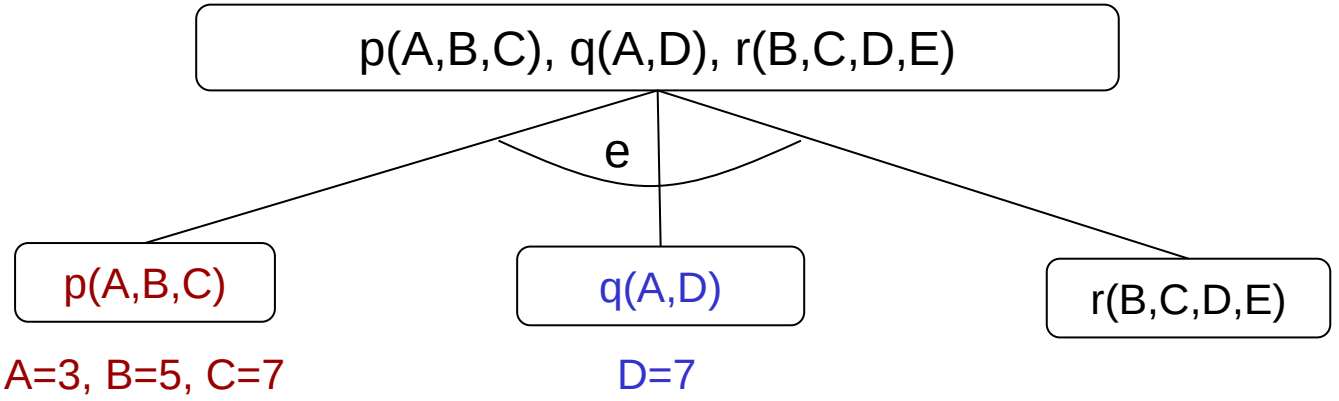


A=3, B=5, C=7

	p(1,2,3).	q(5,10).	r(1,2,3,4).
→	p(3,5,7).	×→ q(1,20).	r(5,7,7,9).
	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

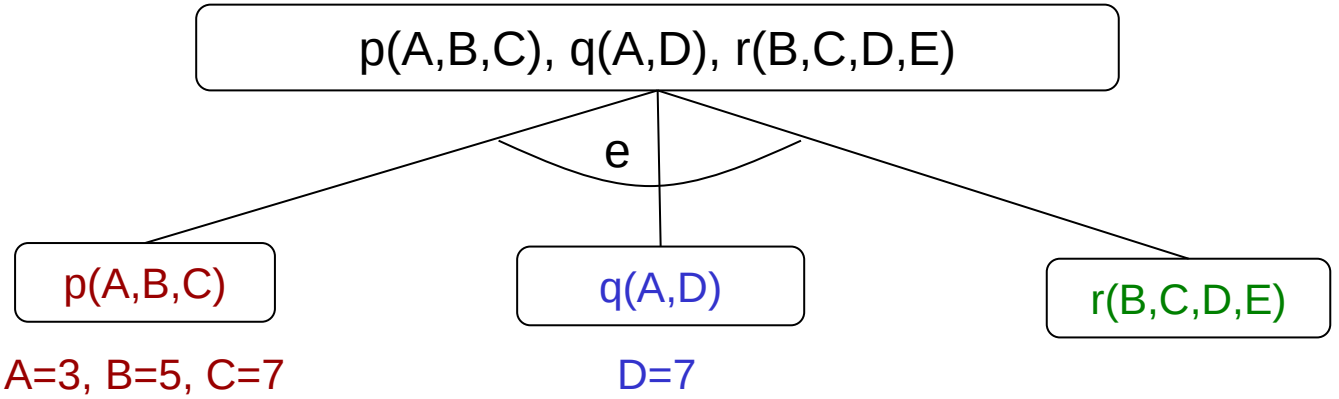
Backtracking



	p(1,2,3).	q(5,10).	r(1,2,3,4).
→	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	→ q(3,7).	r(2,4,6,8).

Variáveis Livres: E

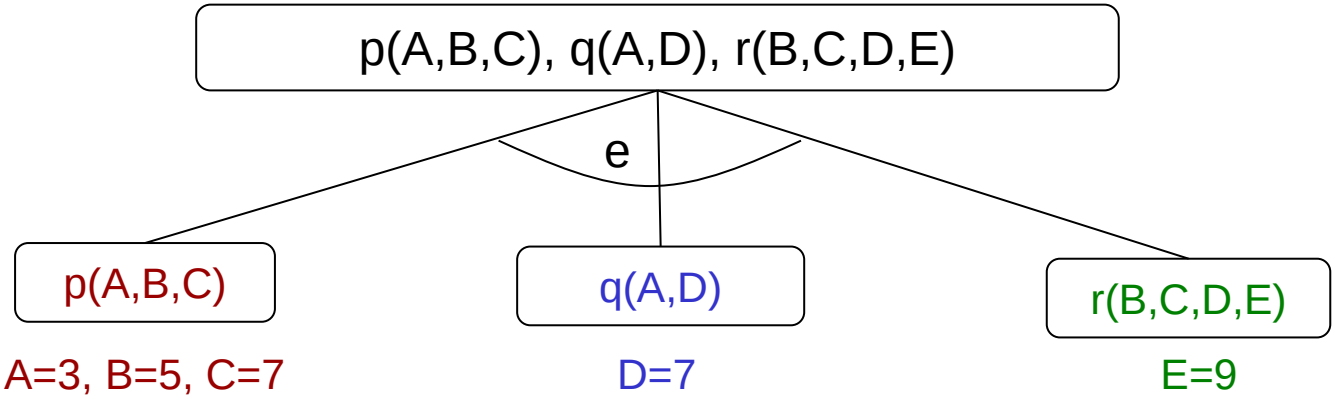
Backtracking



	p(1,2,3).		q(5,10).	× → r(1,2,3,4).
→	p(3,5,7).		q(1,20).	r(5,7,7,9).
	p(2,4,6).	→	q(3,7).	r(2,4,6,8).

Variáveis Livres: E

Backtracking

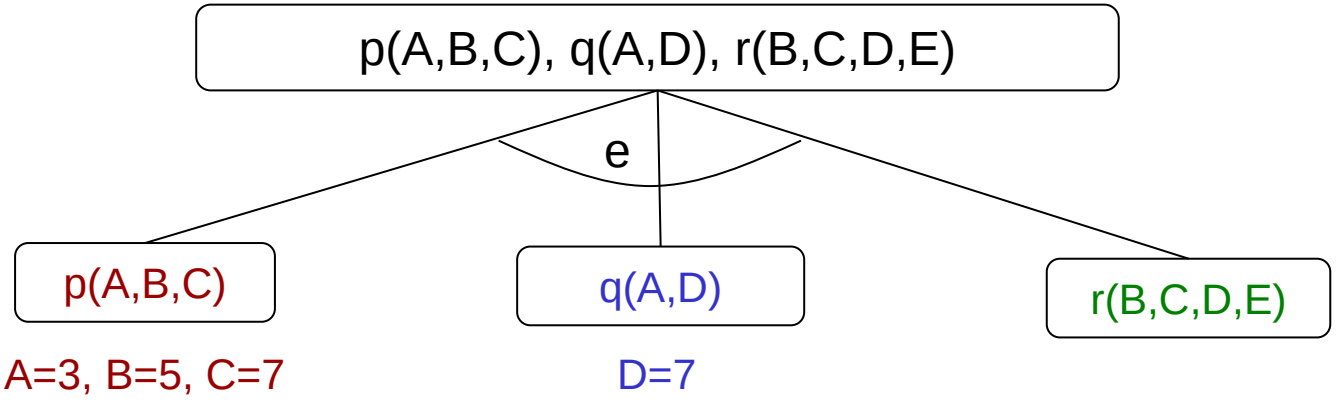


	p(1,2,3).		q(5,10).		r(1,2,3,4).
→	p(3,5,7).		q(1,20).	→	r(5,7,7,9).
	p(2,4,6).	→	q(3,7).		r(2,4,6,8).

Variáveis Livres: nenhuma

Neste caso,
uma solução foi
encontrada:
A=3, B=5, C=7, D=7, E=9

Backtracking

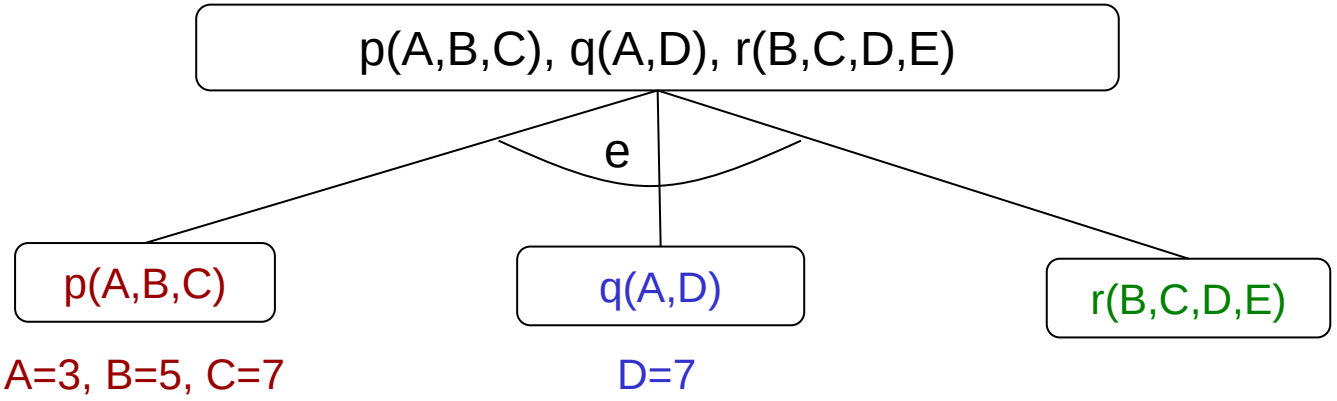


	p(1,2,3).		q(5,10).		r(1,2,3,4).
→	p(3,5,7).		q(1,20).	→	r(5,7,7,9).
	p(2,4,6).	→	q(3,7).		r(2,4,6,8).

Variáveis Livres: E

Caso o usuário solicite outra resposta (digitando **;**), o processo de prova reinicia a partir da situação atual: As variáveis instanciadas na cláusula atual (E) tornam-se variáveis livres novamente

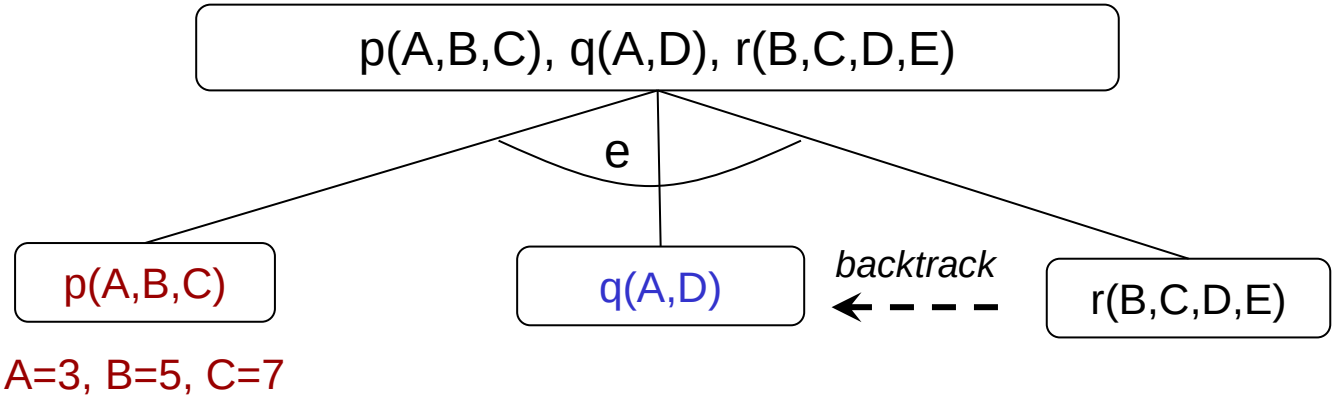
Backtracking



	p(1,2,3).	q(5,10).	r(1,2,3,4).
→	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	→ q(3,7).	×→ r(2,4,6,8).

Variáveis Livres: E

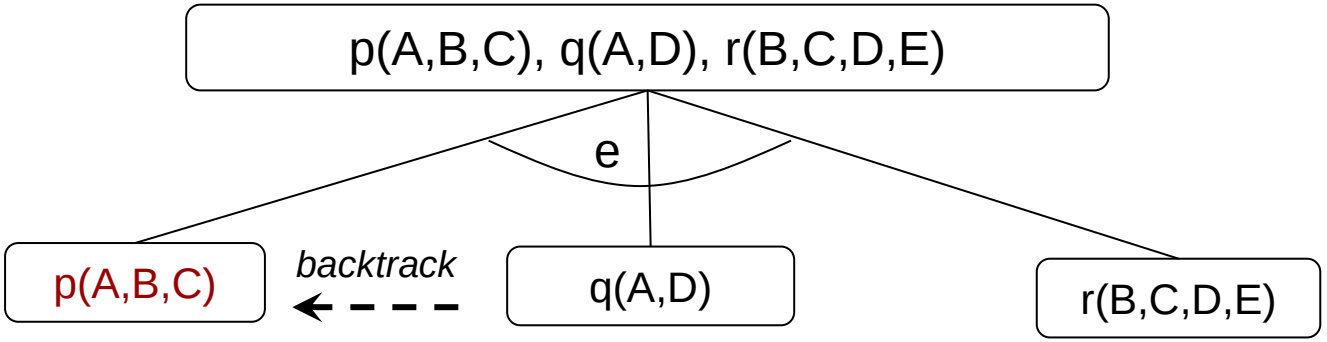
Backtracking



	p(1,2,3).	q(5,10).	r(1,2,3,4).
→	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	→ q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

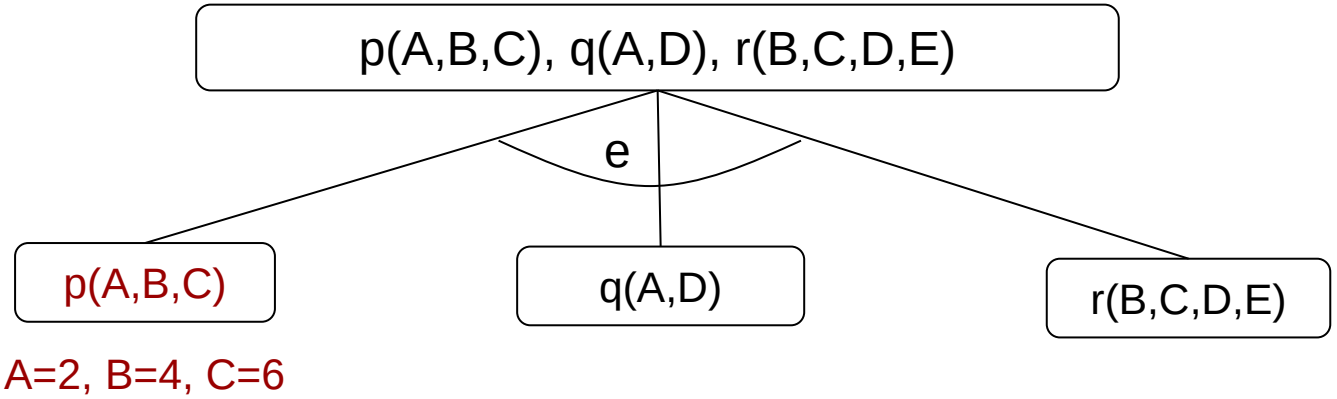
Backtracking



	p(1,2,3).	q(5,10).	r(1,2,3,4).
→	p(3,5,7).	q(1,20).	r(5,7,7,9).
	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: A, B, C, D, E

Backtracking

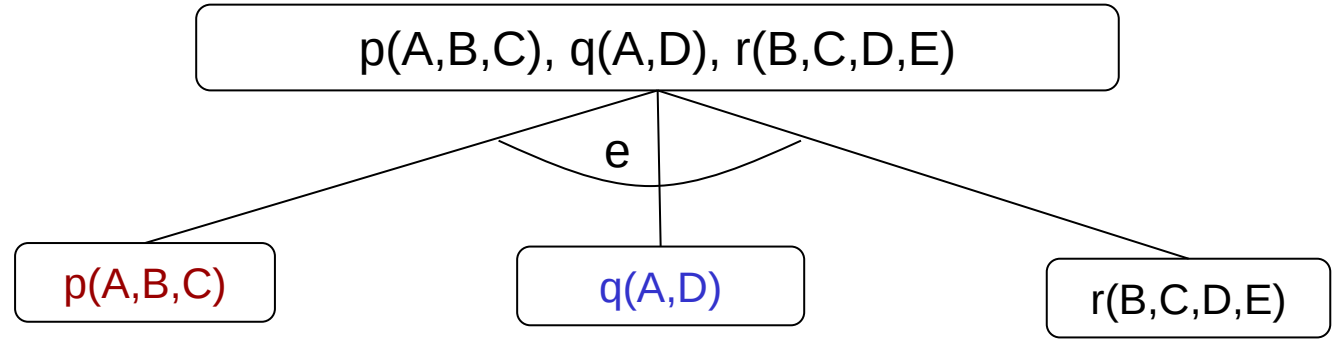


A=2, B=4, C=6

	p(1,2,3).	q(5,10).	r(1,2,3,4).
	p(3,5,7).	q(1,20).	r(5,7,7,9).
→	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

Backtracking

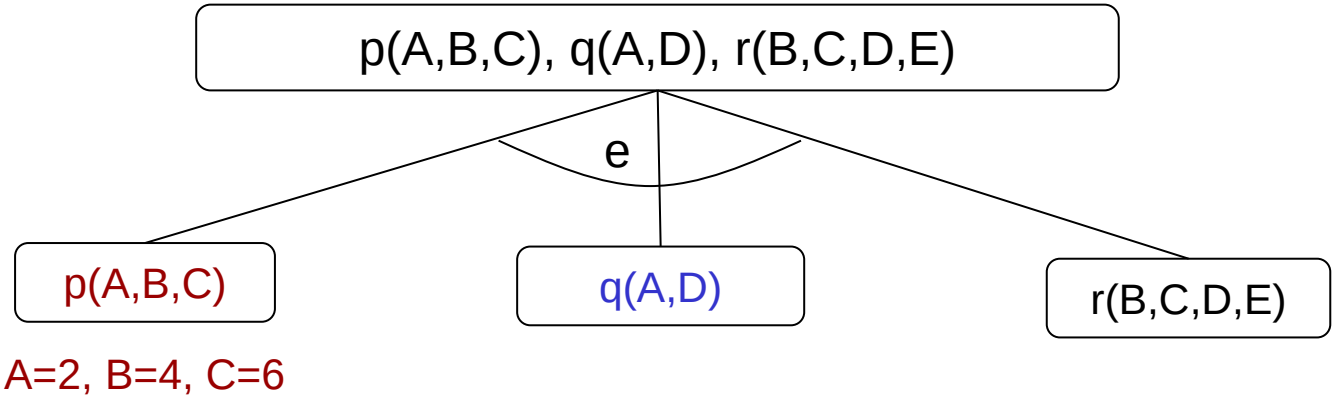


A=2, B=4, C=6

	p(1,2,3).	× → q(5,10).	r(1,2,3,4).
	p(3,5,7).	q(1,20).	r(5,7,7,9).
→	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

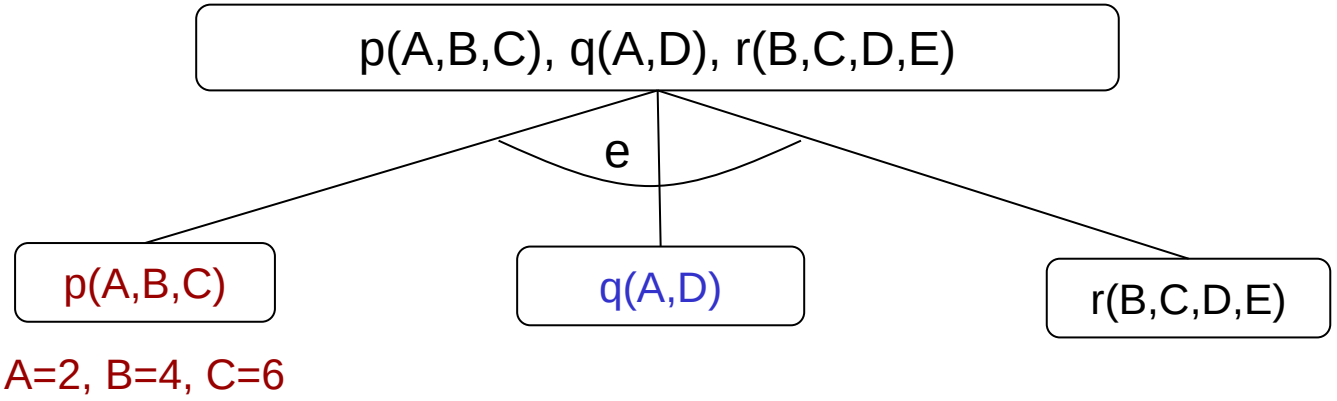
Backtracking



	p(1,2,3).	q(5,10).	r(1,2,3,4).
	p(3,5,7).	✗→ q(1,20).	r(5,7,7,9).
→	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

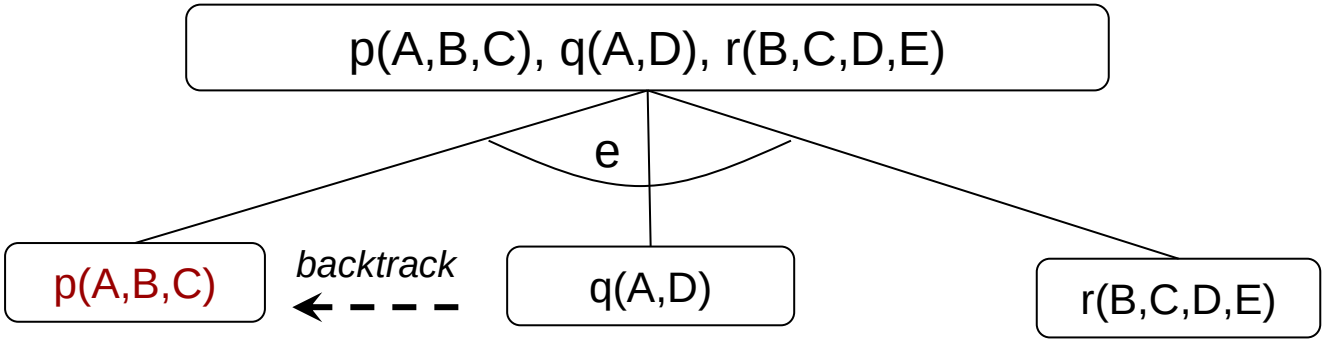
Backtracking



	p(1,2,3).	q(5,10).	r(1,2,3,4).
	p(3,5,7).	q(1,20).	r(5,7,7,9).
→	p(2,4,6).	×→ q(3,7).	r(2,4,6,8).

Variáveis Livres: D, E

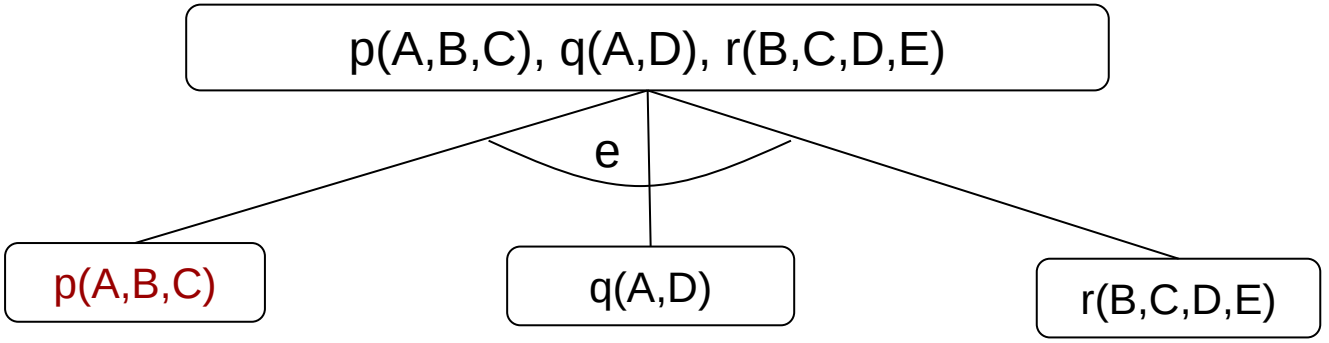
Backtracking



	p(1,2,3).	q(5,10).	r(1,2,3,4).
	p(3,5,7).	q(1,20).	r(5,7,7,9).
→	p(2,4,6).	q(3,7).	r(2,4,6,8).

Variáveis Livres: A, B, C, D, E

Backtracking



p(1,2,3).
p(3,5,7).
p(2,4,6).

q(5,10).
q(1,20).
q(3,7).

r(1,2,3,4).
r(5,7,7,9).
r(2,4,6,8).

Término do processo de prova

Variáveis Livres: A, B, C, D, E

Controlando Backtracking

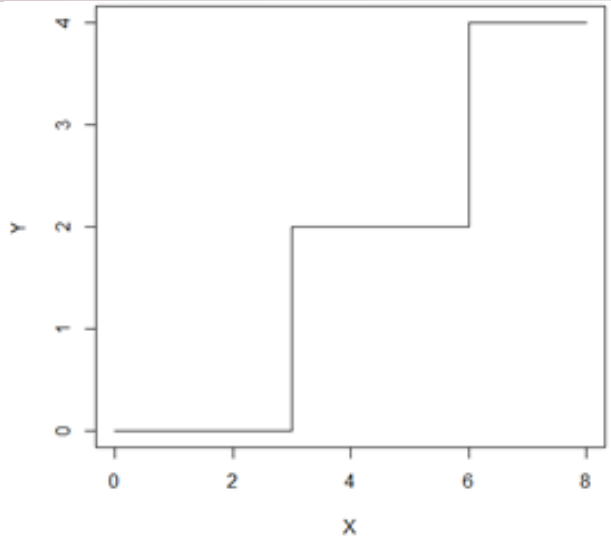
- Mecanismo especial que permite informar Prolog que soluções anteriores não precisam ser reconsideradas quando ele volta (*backtrack*) : o **corte** (*cut*)

- Há duas razões para o uso do corte:
  1. O programa torna-se mais rápido, já que Prolog não gasta tempo tentando satisfazer metas que o programador sabe previamente que nunca vão contribuir para uma solução
  2. O programa ocupa menos memória, uma vez que pontos de *backtracking* não precisam ser mantidos para avaliação futura

- Há dois tipos de cortes
  - **Verde**: caso removido, as soluções encontradas não se alteram (corte foi utilizado apenas por motivos de eficiência)
  - **Vermelho**: caso removido, o programa não apresenta as mesmas soluções; pelo contrário, passa a apresentar um comportamento anormal

Controlando Backtracking

- Considere a função escada
  - Regra 1: se X < 3 então Y = 0
  - Regra 2: se 3 ≤ X < 6 então Y = 2
  - Regra 3: se X ≥ 6 então Y = 4



- Podemos escrever em Prolog como `f(X,Y)`

```
% f(+,?)
f(X,0) :- X < 3.            % Regra 1
f(X,2) :- 3 =< X, X < 6.    % Regra 2
f(X,4) :- X >= 6.           % Regra 3
```

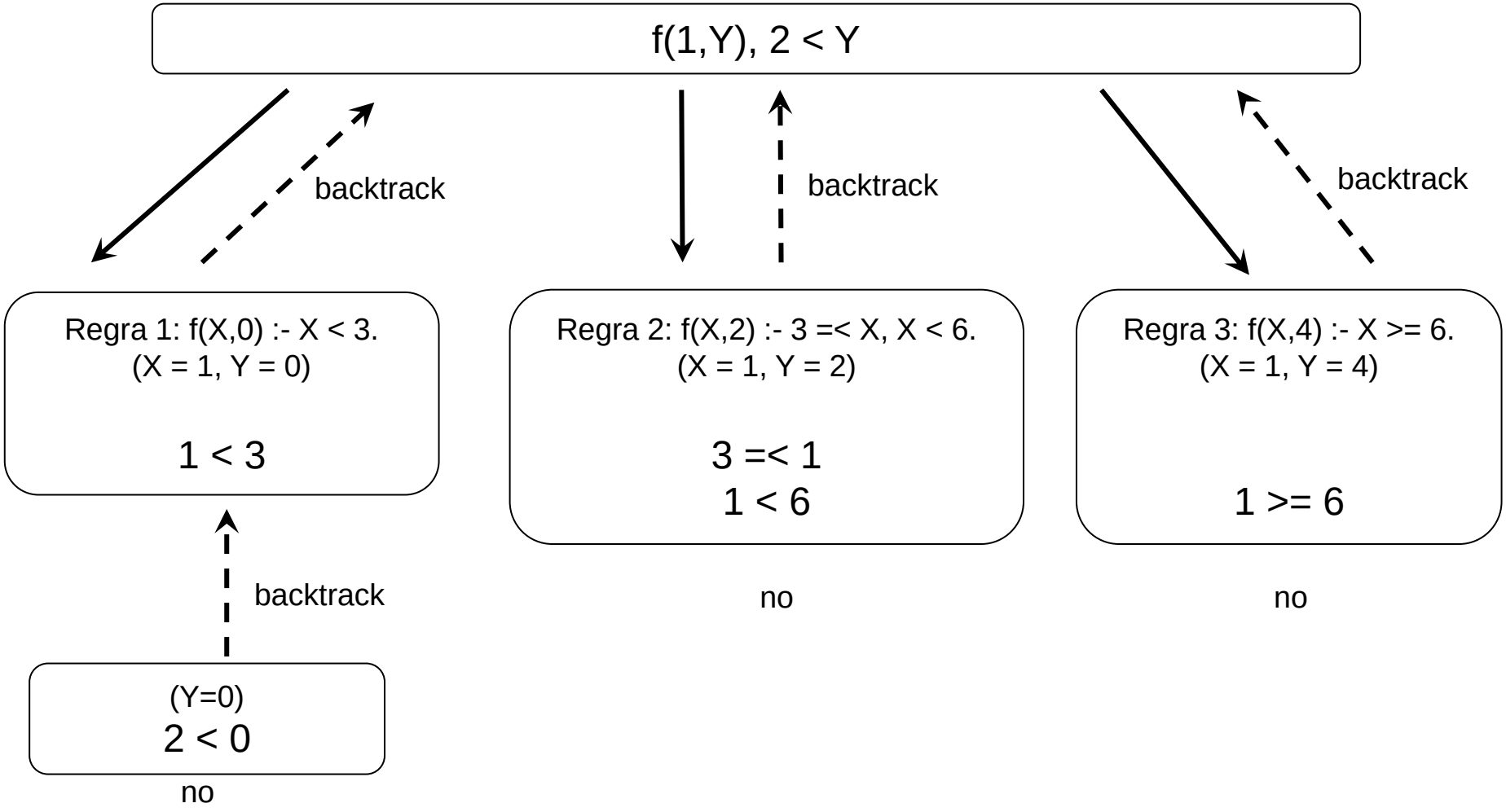
Controlando Backtracking

```
?- f(1,Y), 2<Y.

   f(X,0) :- X < 3.            % Regra 1
   f(X,2) :- 3 =< X, X < 6.    % Regra 2
   f(X,4) :- X >= 6.           % Regra 3
```

- Ao executar a primeira meta, `f(1,Y)`, `Y` é instanciado com 0
- A segunda meta torna-se: 2<0 que falha e assim a pergunta falha
- Mas antes de admitir que a pergunta falhou, Prolog tentou, através de *backtracking*, duas alternativas improdutivas

Controlando Backtracking



Controlando Backtracking

- As três regras sobre a relação **f** são mutuamente exclusivas, assim, no máximo, uma delas será verdadeira
- Portanto, o programador (ao invés do Prolog) sabe que tão logo uma das regras tenha sucesso, então não há razão para continuar tentando utilizar as demais regras, pois vão falhar
- No exemplo, sabe-se que a Regra 1 teve sucesso
- De modo a evitar *backtracking* inútil o programador deve informar explicitamente para Prolog não realizar *backtrack*, através do corte
- O corte é escrito como ! e inserido entre as metas

Controlando Backtracking

- O programa com corte **verde** fica assim:

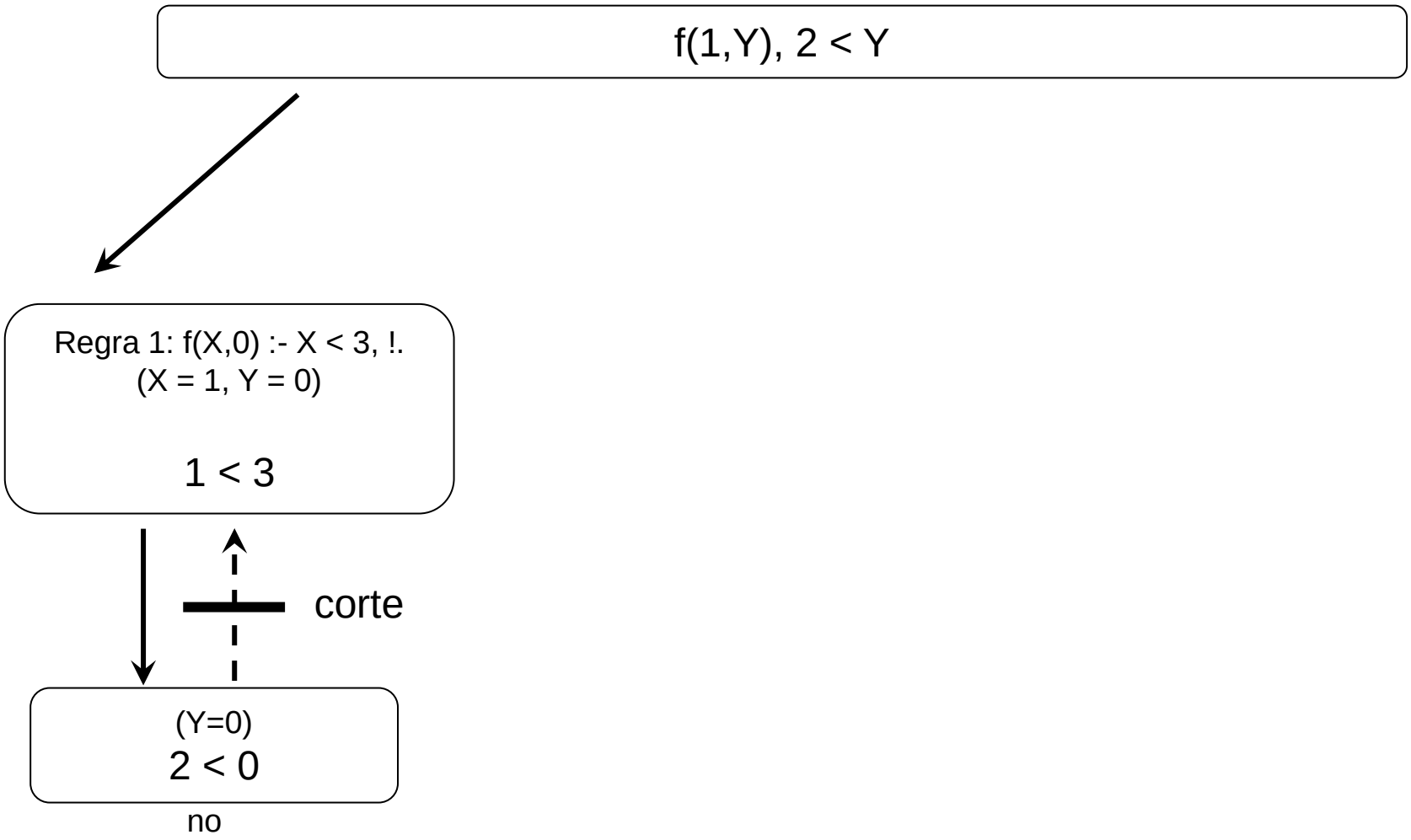
```
f(X,0) :- X < 3, !.          % Regra 1
f(X,2) :- 3 =< X, X < 6, !. % Regra 2
f(X,4) :- X >= 6.            % Regra 3
```

- O símbolo **!** evita *backtracking* nos pontos indicados, como demonstrado em breve

Controlando Backtracking

- Se fizermos a mesma pergunta
  ```
  ?- f(1,Y), 2<Y.
  ```
  - Com os cortes, os ramos alternativos que correspondem às Regras 2 e 3 não serão gerados
  - Este programa com corte é mais eficiente que a versão original: quando a execução falha, este programa irá reconhecer este fato, mais cedo que o programa original
- Os cortes são **verdes** neste exemplo, pois se eles forem removidos, o programa ainda produzirá os mesmos resultados, mas gastará mais tempo

Controlando Backtracking



Controlando Backtracking

- Ainda considerando:

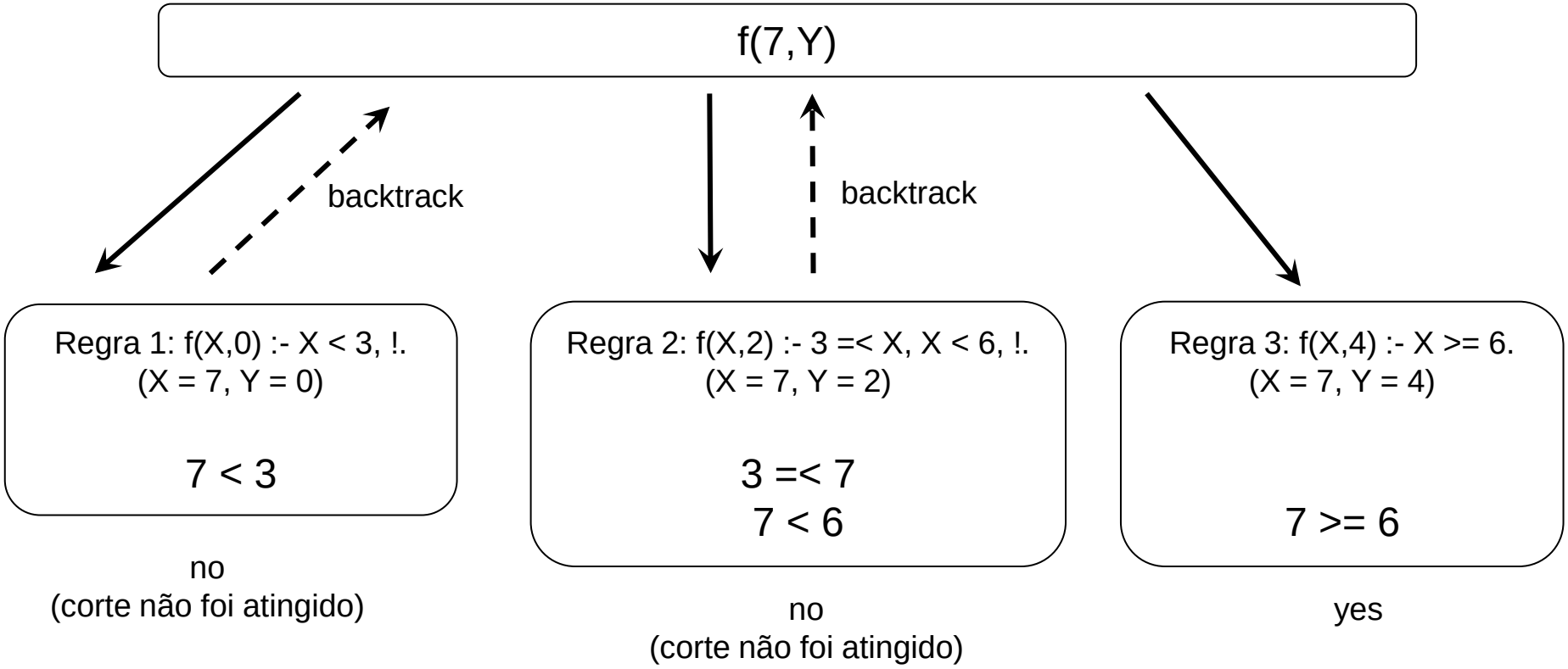
```
f(X,0) :- X < 3, !.          % Regra 1
f(X,2) :- 3 =< X, X < 6, !. % Regra 2
f(X,4) :- X >= 6.            % Regra 3
```

- Suponha agora a seguinte pergunta

```
?- f(7,Y).
Y = 4
```

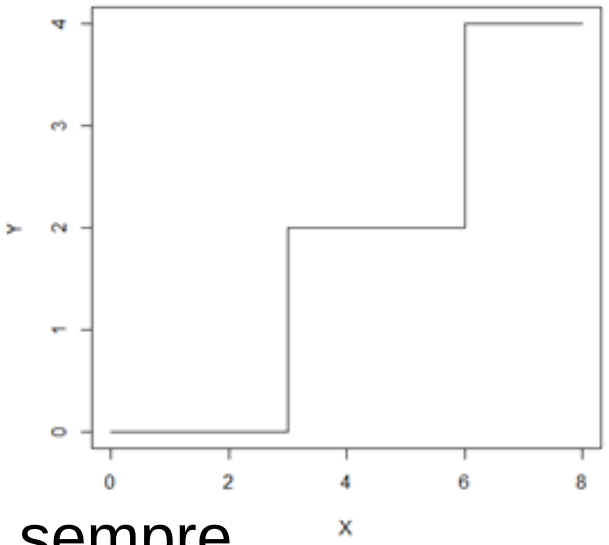
- Note que todas as 3 regras são avaliadas antes que uma resposta seja encontrada
  - Tente Regra 1: 7 < 3 falha, *backtrack* e tente Regra 2 (corte não foi atingido, pois falha na primeira cláusula e uso do operador E já implica que Regra 1 falhou)
  - Tente Regra 2: 3 ≤ 7 é satisfeita mas 7 < 6 falha, *backtrack* e tente Regra 3 (novamente corte não foi atingido)
  - Tente Regra 3: 7 ≥ 6 é satisfeita

Controlando Backtracking



Controlando Backtracking

- Isto mostra outra fonte de ineficiência:
  - Primeiro é estabelecido que X<3 não é verdadeiro (7<3 falha)
  - A seguir, a cláusula 3=<X (3=<7) é satisfeita, mas sabemos que uma vez que o primeiro teste (X<3) falhou, o segundo teste (3=<X) será satisfeito, pois nega o primeiro
  - Portanto, o segundo teste é redundante e pode ser omitido
  - O mesmo ocorre com X >= 6 na Regra 3



- Isto leva à seguinte formulação:
  - Se X < 3 então Y = 0
  - Senão Se X < 6 então Y = 2
  - Senão Y = 4

- Podemos omitir as condições que são sempre verdadeiras quando executadas

Controlando Backtracking

- Isto nos leva à terceira versão, com corte **vermelho**:

```
f(X,0) :- X < 3, !.          % Regra 1
f(X,2) :- X < 6, !.          % Regra 2
f(X,4).                      % Regra 3
```

- O programa com corte vermelho produz as mesmas soluções que a versão original, mas é mais eficiente que as duas versões anteriores

Controlando Backtracking

- Entretanto, o que ocorre se agora removermos os cortes?

```
f(X,0) :- X < 3.          % Regra 1
f(X,2) :- X < 6.          % Regra 2
f(X,4).                   % Regra 3
```

- O programa agora produz várias soluções, algumas das quais incorretas, por exemplo:
  - `?- f(1,Y).`
  - **`Y = 0;`**
  - **`Y = 2;`**
  - **`Y = 4`**

Controlando Backtracking

O mecanismo de corte funciona do seguinte modo:

- Seja meta-pai, a meta que unificou com a cabeça da cláusula contendo o corte
- Quando o corte é encontrado, como uma meta ele é satisfeito imediatamente mas ele estabelece um compromisso de manter todas as escolhas efetuadas entre o momento em que a meta-pai foi invocada e o momento no qual o corte foi encontrado
- Todas as alternativas restantes entre a meta-pai e o corte são descartadas

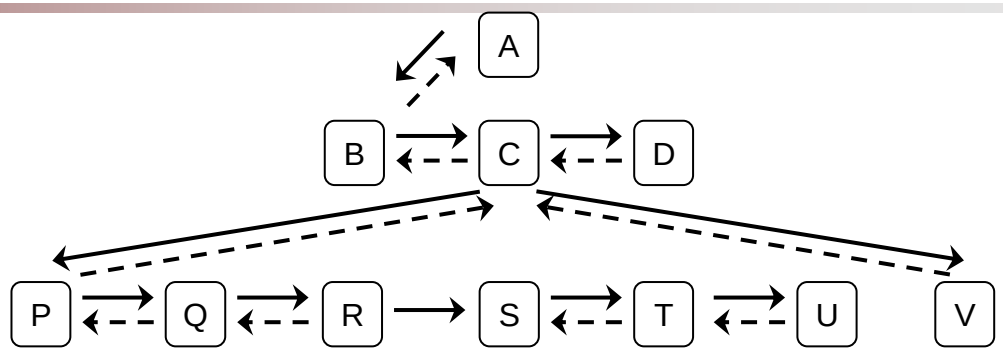
Controlando Backtracking

- Para ilustrar o mecanismo de corte, considere a cláusula
  - `H :- B1, B2, ..., Bm, !, ..., Bn.`
- Suponha que uma meta G unificou com H (nesse caso, G é a meta pai)
- No momento em que o corte é encontrado, o sistema já encontrou alguma solução para as metas `B1, ..., Bm`.
- Quando o corte é executado, esta solução atual `B1, ..., Bm` é congelada e todas alternativas possíveis são descartadas
- Além disso, a meta G fica comprometida a esta cláusula: qualquer tentativa de unificar G com a cabeça de uma outra cláusula é excluída devido ao mecanismo do corte

Controlando Backtracking

- Por exemplo

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
?- A.
```



- O corte afetará a execução da meta C
- *Backtracking* será possível dentro da lista de metas P, Q, R ; entretanto, tão logo o corte seja encontrado, todas as soluções alternativas de P, Q, R são descartadas
- Ao encontrar o corte, também será descartada a cláusula alternativa para a meta C

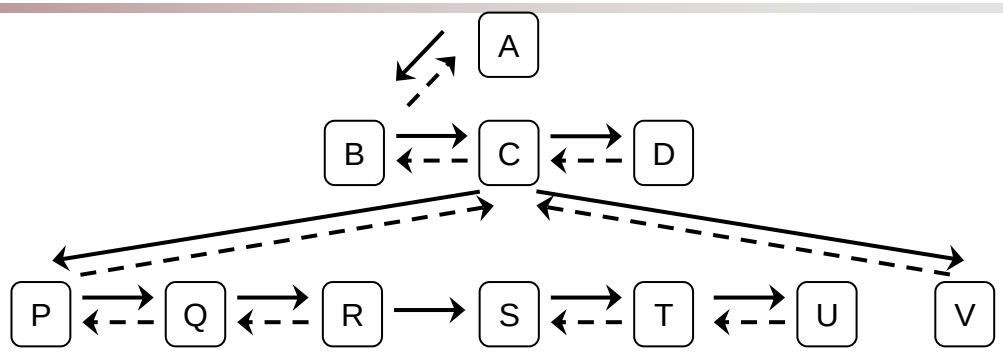
```
C :- V.
```

- Entretanto, *backtracking* ainda será possível entre a lista de metas S, T, U

Controlando Backtracking

- Por exemplo

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
?- A.
```



- A meta-pai da cláusula contendo o corte é a meta C (na árvore e) na cláusula
  - `A :- B, C, D.`
- Portanto, o corte afetará apenas a execução da meta C; mas será ‘invisível’ de dentro da meta A: *backtracking* automático continuará a existir entre a lista de metas B, C, D apesar do corte dentro da cláusula usada para satisfazer C

Negação por Falha

Exemplo 1: Suponha os seguintes fatos (BC):

gosta(joao,peixe).

gosta(joao,maria).

gosta(maria,livro).

gosta(pedro,livro).

```
?- gosta(joao,dinheiro).
no
?- gosta(maria,joao).
no
?- gosta(maria,livro).
yes
?- rei(joao,inglaterra).
no
```

Negação por Falha

```
?- rei(joao,inglaterra).
no
```

Como não exista nenhum fato na base sobre a relação “rei”, Prolog responde “no” (ou “false”) significando que nada foi encontrado que pudesse provar o que foi pedido

- “no” não é necessariamente o oposto da verdade nesse caso
- “no” significa “não foi possível provar”

Negação por Falha

- Exemplo 2:

```
?- humana(maria).
no
?- \+ humana(maria).
yes
```

- A segunda resposta não deve ser entendida como ‘Maria não é humana’
- O que Prolog quer dizer é: ‘Não há informação suficiente no programa para provar que Maria é humana’
- Isto ocorre porque Prolog não tenta provar diretamente `\+humana(maria)`, mas sim o seu oposto `humana(maria)`. Se o oposto não pode ser provado, Prolog assume que a negação `\+humana(maria)` é verdadeira

Negação por Falha

- O predicado \+ é definido como:

```
\+(X) :-
    X,
    !,
    fail.
\+(X).
```

- Ou, alternativamente, como:

```
\+(X) :-
    X,
    !,
    fail;
    true.
```

- Que é equivalente a:

```
\+(X) :-
    (X, !, fail);
    true.
```

Negação por Falha

- Prolog trabalha com o conceito de mundo fechado

- O mundo é fechado no sentido que tudo o que existe está declarado no programa ou pode ser derivado do programa

- Assim, se algo não está no programa (ou não pode ser derivado dele) então ele é falso e, conseqüentemente, sua negação é verdadeira

Negação por Falha

- Exemplo 3:

```
bom_padrao(jeanluis).
caro(jeanluis).
bom_padrao(francesco).
razoavel(Restaurante) :-
    \+ caro(Restaurante).

?- bom_padrao(X),
   razoavel(X)
X = francesco

?- razoavel(X),
   bom_padrao(X).
no
```

- A diferença entre ambas perguntas é que a variável `X` está, no primeiro caso, já instanciada quando `razoavel(X)` é executado, ao passo que `X` não está instanciada no segundo caso
- A dica geral é `\+ Meta` funciona com segurança se as variáveis em `Meta` estão todas instanciadas no momento em que `\+ Meta` é chamada; caso contrário pode-se obter resultados inesperados

Negação por Falha

- Há diferença entre?
  - `?- pertence(X,[a,b,c]).`
  - `?- \+ \+ pertence(X,[a,b,c]).`
- Há uma tendência a dizer que não, por pensar do seguinte modo:
  - `pertence(X,[a,b,c])` tem sucesso então
  - `\+ pertence(X,[a,b,c])` falha, portanto
  - `\+ \+ pertence(X,[a,b,c])` tem sucesso
- Isto está apenas em parte correto

Negação por Falha

Isto é o que ocorre: `?- \+ \+ pertence(X,[a,b,c]).`

- `pertence(X,[a,b,c])` é verdadeiro e instancia `X=a`
- Prolog tenta provar `\+ pertence(X,[a,b,c])` e falha (pois conseguiu provar o oposto)
- Quando uma cláusula falha, qualquer variável que se tornou instanciada através daquela cláusula torna-se livre
- Assim `X` torna-se uma variável livre
- Prolog tenta provar `\+ \+ pertence(X,[a,b,c])` e tem sucesso, mas `X` é uma variável livre
- Portanto, Prolog escreve a variável livre

```
?- pertence(X,[a,b,c]).
X = a;
X = b;
X = c
?- \+ \+ pertence(X,[a,b,c]).
X = _G390
```

Predicados de Entrada e Saída

Estes predicados têm sucesso (são verdadeiros) uma única vez (não podem ser re-satisfeitos através de *backtracking*)

- **get_char(X)**: é verdadeiro se X unifica com o próximo caractere encontrado na entrada atual
- **put_char(X)**: escreve o caractere X na saída atual (um erro ocorre se X está livre); se X é número, ele é interpretado como posição de caractere conforme uma codificação – exemplo: tabela ASCII
- **read(X)**: lê um termo (objeto) da entrada e é verdadeiro se o termo lido unifica com X; o termo deve ser terminado por um ponto-final '**.**' que não faz parte do termo lido
- **write(X)**: escreve o termo X na saída; variáveis livres são numeradas de forma única e geralmente iniciam com um *underscore* '_'
- **nl**: escreve um caractere de nova linha (*new line*)
- **tab(N)**: escreve N espaços em branco

Predicados de Leitura de Arquivos

Estes predicados têm sucesso (são verdadeiros) uma única vez (não podem ser re-satisfeitos através de *backtracking*)

- **see(F):** direciona a entrada de dados para o arquivo F, o qual pode conter uma base de fatos e regras
- **tell(F):** direciona a saída de dados para o arquivo F
- **read(X):** lê uma linha da entrada atual (um arquivo, se ele estiver aberto para leitura) e coloca em X
- **assert(X):** coloca (escreve) o predicado X na base de fatos e regras
- **seen(F):** fecha o arquivo F aberto para leitura
- **told(F):** fecha o arquivo F aberto para escrita
- **end_of_file:** constante de fim de arquivo

Exemplo

- Usando um programa com `pertence/2`, suponha que desejamos imprimir todas as soluções encontradas, sem precisar digitar '**;**'
- Podemos estender o programa do seguinte modo:

```
pertence(E, [E|_]).
pertence(E, [_|T]) :-
        pertence(E,T).
mostra_todos(Lista) :-
     pertence(X,Lista),
     write('elemento '),
     write(X),
     nl,
     fail.
```

```
?- mostra_todos([a,b,c]).
elemento a
elemento b
elemento c
no
```

Exemplo

- Embora o programa tenha impresso todos os elementos da lista, ele falha
- Uma forma de garantir que o programa tenha sucesso ao terminar é dada na versão ao lado

```
pertence(E, [E|_]).
pertence(E, [_|T]) :-
        pertence(E,T).
mostra_todos(Lista) :-
     pertence(X,Lista),
     write('elemento '),
     write(X),
     nl,
     fail.
mostra_todos(_).

?- mostra_todos([a,b,c]).
elemento a
elemento b
elemento c
yes
```

Predicados Adicionais

- **findall(Vars,Meta,L)** constrói uma lista L consistindo de todos os objetos Vars tais que a Meta é satisfeita
  - As variáveis não declaradas em Vars que estão em Meta são consideradas existencialmente quantificadas (“já existem”)
- Exemplo

```
idade(pedro,7).
idade(ana,5).
idade(alice,8).
idade(tomaz,5).
?- findall(Crianca,idade(Crianca,Idade),L).
L = [pedro,ana,alice,tomaz]
?- findall([Crianca,Idade],idade(Crianca,Idade),L).
L = [[pedro,7], [ana,5], [alice,8], [tomaz,5]]
?- findall([Crianca,Idade],
           (idade(Crianca,Idade),Idade>5),L).
L = [[pedro,7], [alice,8]]
```

Predicados Adicionais

- **bagof(Vars,Meta,L)** constrói uma lista L consistindo de todos os objetos Vars tais que a Meta é satisfeita
  - As variáveis não declaradas em Vars que estão em Meta **não** são consideradas existencialmente quantificadas (“não existem”). Devem ser declaradas explicitamente na Meta
- Exemplo

```
idade(pedro,7).
idade(ana,5).
idade(alice,8).
idade(tomaz,5).
?- bagof(Crianca,idade(Crianca,Idade),L).
Idade = 5,
L = [ana, tomaz];
Idade = 7,
L = [pedro];
Idade = 8,
L = [alice].
```

Predicados Adicionais

- **bagof(Vars,Meta,L)** constrói uma lista L consistindo de todos os objetos Vars tais que a Meta é satisfeita
  - As variáveis não declaradas em Vars que estão em Meta **não** são consideradas existencialmente quantificadas (“não existem”). Devem ser declaradas explicitamente na Meta
- Exemplo

```
idade(pedro,7).
idade(ana,5).
idade(alice,8).
idade(tomaz,5).
?- bagof(Crianca,Idade^idade(Crianca,Idade),L).
L = [pedro, ana, alice, tomaz].
```

Predicados Adicionais

- **setof(Vars,Meta,L)** constrói uma lista L consistindo de todos os objetos Vars tais que a Meta é satisfeita
  - As variáveis não declaradas em Vars que estão em Meta **não** são consideradas existencialmente quantificadas (“não existem”). Devem ser declaradas explicitamente na Meta
  - Ordena e elimina repetições
- Exemplo

```
idade(pedro,7).
idade(ana,5).
idade(alice,8).
idade(tomaz,5).
?- setof(Crianca,Idade^idade(Crianca,Idade),L).
L = [alice, ana, pedro, tomaz].
```

Slides baseados em:

Bratko, I.;
Prolog Programming for Artificial Intelligence,
3rd Edition, Pearson Education, 2001.

Clocksin, W.F.; Mellish, C.S.;
Programming in Prolog,
5th Edition, Springer-Verlag, 2003.

Programas Prolog para o
Processamento de Listas e Aplicações,
Monard, M.C & Nicoletti, M.C., ICMC-USP, 1993

Material elaborado por José Augusto Baranauskas
Adaptado por Huei Diana Lee