Backtracking

- A evolução de busca por soluções no Prolog assume a forma de uma árvore de busca.
  - Percorrida de cima para baixo da esquerda para a direita (busca em profundidade)

Backtracking

- Os objetivos em um programa Prolog podem ser bem-sucedidos ou falhar
  - Objetivo bem-sucedido: deve ser unificado com a cabeça de uma cláusula do programa e todos os objetivos no corpo desta cláusula devem também ser bem-sucedidos
  - Se tais condições não ocorrerem, então o objetivo falha
    - Falha em um nodo terminal da árvore de pesquisa: o Prolog aciona o mecanismo de *backtracking*, retornando pelo mesmo caminho percorrido (encontrar soluções alternativas)

Backtracking

- Exemplo

```
% gosta(X, Y)  "X gosta de Y"
gosta(joão, jazz).
gosta(joão, renata).
gosta(joão, lasanha).
gosta(renata, joão).
gosta(renata, lasanha).

?- gosta(joão, X), gosta(renata, X).
```

Backtracking

- O *backtracking* automático é uma ferramenta muito poderosa e a sua exploração é de grande utilidade para o programador.
  - Entretanto, ele pode se transformar em fonte de ineficiência
  - Solução: mecanismo de “poda” da árvore de busca

O operador "*cut*"

- O predicado de corte !/0 oferece um modo de controlar o retrocesso.

- Razões para seu uso
  I. O programa irá executar mais rapidamente, porque não irá desperdiçar tempo tentando satisfazer objetivos que não irão contribuir para a solução desejada
  II. A memória será economizada, uma vez que determinados pontos de *backtracking* não necessitam ser armazenados para exame posterior

O operador “*cut*”

- Sintaticamente o *cut* em uma cláusula tem a aparência de um objetivo sem nenhum argumento, representado por um ponto de exclamação "!“

Aplicações do *cut*

- Unificação de padrões, de forma que quando um padrão é encontrado os outros padrões possíveis são descartados
- Na implementação da negação como regra de falha
- Para eliminar da árvore de pesquisa soluções alternativas quando uma só é suficiente
- Para encerrar a pesquisa quando a continuação iria conduzir a uma pesquisa infinita, etc.

Exemplo 1: código sem corte

- Problema

(1)Se X < 3, então Y = 0

(2)Se 3 ≤ X e X < 6, então Y = 2

(3)Se 6 ≤ X, então Y = 4

- Solução em Prolog

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

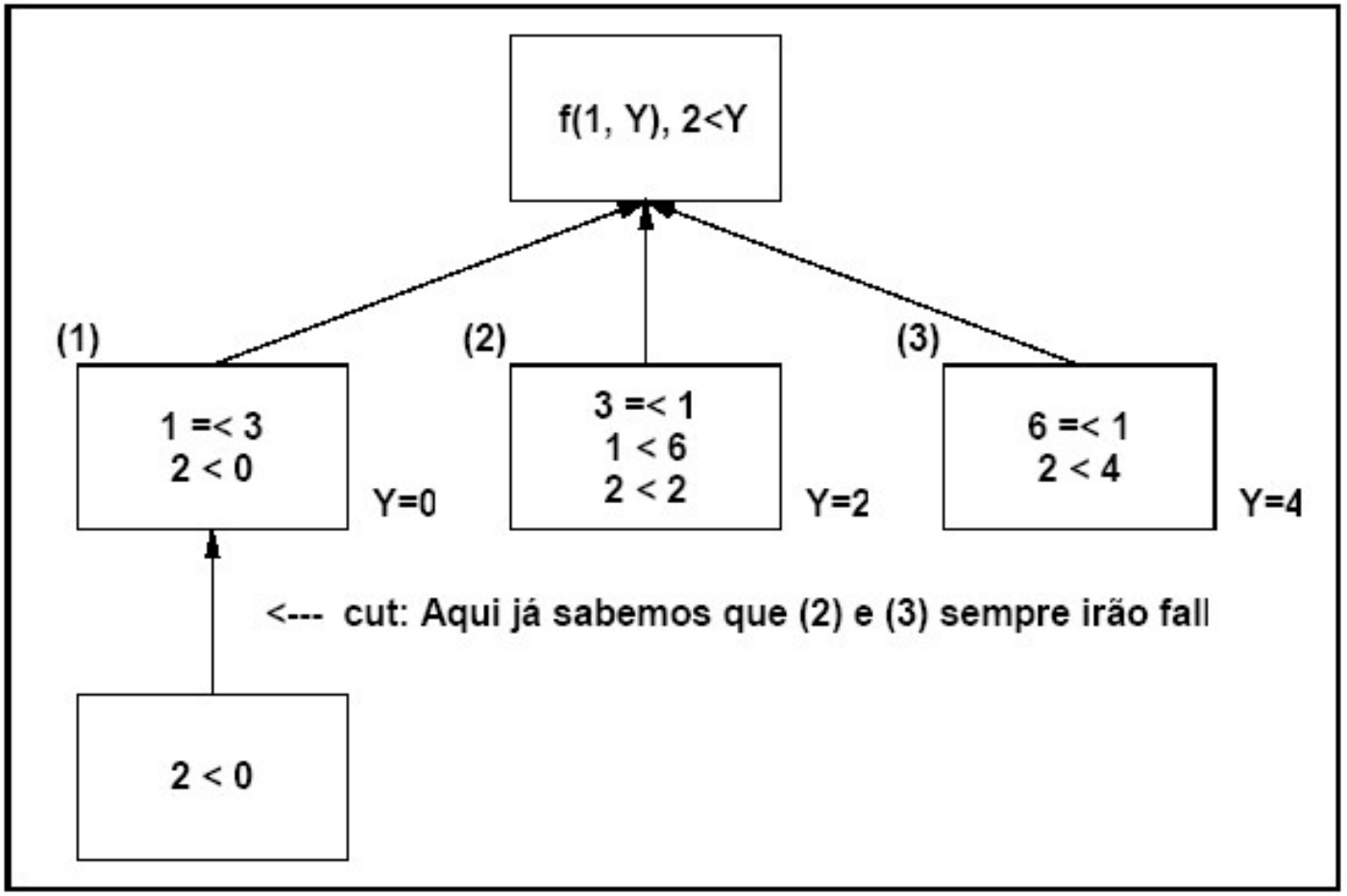
Exemplo 1: código sem corte

- Exclusão Mútua
  - Pergunta

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

  - Solução
    - Y é instanciada com 0 e o objetivo 2 < 0, falha
    - Avaliação das duas outras regras → falha

Exemplo 1



Luis, A. M. Palazzo, Introdução à Programação Prolog, Educat, 1997.

Exemplo 1: código com corte

- Programa reescrito com o uso do cut:

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

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

Nesse caso, a introdução de *cuts* afetou somente a interpretação operacional

O que o corte faz?

- O corte somente restringe-nos às escolhas já feitas desde que o objetivo pai foi unificado com o lado esquerdo da cláusula contendo o corte.
- Por exemplo, em uma regra da forma

  q:- p_1, … , p_n, !, r_1, … , r_n.

  Quando alcançarmos o corte ele nos restringe:
  - a esta cláusula particular de **q**
  - às escolhas feitas por **p_1, … , p_n**
  - Mas, **NÃO** às escolhas feitas por **r_1, … , r_n**

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,3).
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,3).
true
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,3).
true

?- max(7,3,7).
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,3).
true

?- max(7,3,7).
true
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,2).
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,2).
false
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,2).
false

?- max(2,3,5).
```

Usando Corte

- Considere o seguinte predicado max/3 que tem sucesso se o terceiro argumento é o maior dos dois primeiros

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

```
?- max(2,3,2).
false

?- max(2,3,5).
false
```

Usando Corte

- Qual é o problema?
- Existe uma ineficiência em potencial
  - Suponha que ele é chamado com ?- max(3,4,Y).
  - Ele corretamente unificará Y com 4
  - Mas, quando lhe é solicitado mais soluções, ele tentará satisfazer a segunda cláusula. E isto é completamente sem sentido!

```
max(X,Y,Y):- X =< Y.
max(X,Y,X):- X>Y.
```

max/3 com corte

- Com a ajuda do corte isto é fácil de consertar

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X):- X>Y.
```

- Note como isto funciona:
  - Se X =< Y sucede, o corte nos restringe a esta escolha e a segunda cláusula de max/3 não é considerada
  - Se X =< Y falha, Prolog segue para a segunda cláusula

Cortes verdes

- Cortes que não alteram o significado de um predicado são chamados de **cortes verdes**
- O corte em max/3 é um exemplo de corte verde:
  - O novo código produz exatamente as mesmas respostas que a versão antiga,
  - mas é mais eficiente!

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X).
```

- Quão bom é isto?

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X).
```

- Quão bom é isto?

```
?- max(200,300,X).
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X).
```

- Quão bom é isto?
  - ok

```
?- max(200,300,X).
X=300
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X).
```

- Quão bom é isto?

```
?- max(400,300,X).
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X).
```

- Quão bom é isto?
  - ok

```
?- max(400,300,X).
X=400
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X).
```

- Quão bom é isto?

```
?- max(200,300,200).
```

Um outro max/3 com corte

- Porque não remover o corpo da segunda cláusula? Obviamente, ela é redundante.

```
max(X,Y,Y):- X =< Y, !.
max(X,Y,X).
```

- Quão bom é isto?
  - ôpa!

```
?- max(200,300,200).
true
```

max/3 revisado com corte

- Unificação após cruzar o corte

```
max(X,Y,Z):- X =< Y, !, Y=Z.
max(X,Y,X).
```

- Isto de fato funciona

```
?- max(200,300,200).
```

max/3 revisado com corte

- Unificação após cruzar o corte

```
max(X,Y,Z):- X =< Y, !, Y=Z.
max(X,Y,X).
```

- Isto de fato funciona

```
?- max(200,300,200).
false
```

Cortes Vermelhos

- Cortes que alteram o significado de um predicado são chamados de **cortes vermelhos**
- O corte no max/3 revisado é um exemplo de corte vermelho:
  - Se retirarmos o corte, nós não obteremos um programa equivalente ao original

- **Programas contendo cortes vermelhos**
  - **Não são completamente declarativos**
  - **Podem ser difíceis de ler**
  - **Podem levar a erros sutis de programação**