INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SÃO PAULO,
CÂMPUS BIRIGUI - SP
BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

ISADORA DISPOSTI BUENO DOS SANTOS

EXERCÍCIOS – PROLOG

BIRIGUI - SP
2023

Capítulo 3 e 4

3.1.

O programa a seguir associa a cada pessoa seu esporte preferido.

```
joga(ana,volei).
joga(bia,tenis).
joga(ivo,basquete).
joga(eva,volei).
joga(leo,tenis).
```

Suponha que desejamos consultar esse programa para encontrar um parceiro P para jogar com Leo. Então, podemos realizar essa consulta de duas formas:

a) ?- joga(P,X), joga(leo,X), P=leo. b) ?- joga(leo,X), joga(P,X), P=leo.

Desenhe as árvores de busca construídas pelo sistema ao responder cada uma dessas consultas. Qual consulta é mais eficiente, por quê?

R: A consulta b) é mais eficiente, pois ela faz a consulta mais restritiva primeiro, ou seja, ela busca primeiro o esporte que o Leo joga e depois busca quem joga esse esporte, enquanto a consulta a) busca primeiro quem joga e depois o esporte que essa pessoa joga.

Arvore de busca da consulta a):

```
?- joga(P,X), joga(leo,X), P\=leo.
    |
    |-- joga(ana, volei) (P = ana, X = volei)
    |   |
    |   |-- joga(leo, volei) (Fail) (Backtrack)
    |
    |-- joga(bia, tenis) (P = bia, X = tenis)
    |   |
    |   |-- joga(leo, tenis) (P = leo, X = tenis)
    |   |   |
    |   |   |-- bia\=leo (true) (P = bia, X = tenis) (Print)
    |
    |-- joga(ivo, basquete) (P = ivo, X = basquete)
    |   |
    |   |-- joga(leo, basquete) (Fail) (Backtrack)
    |
    |-- joga(eva, volei) (P = eva, X = volei)
    |   |
    |   |-- joga(leo, volei) (Fail) (Backtrack)
    |
    |-- joga(leo, tenis) (P = leo, X = tenis)
```

```
            |
            |-- leo\=leo (Fail) (Backtrack)
```

Arvore de busca da consulta b):

```
    ?- joga(leo, X), joga(P, X), P\=leo.
        |
        |-- joga(leo, tenis) (X = tenis)
        |   |
        |   |-- joga(bia, tenis) (P = bia)
        |   |   |
        |   |   |-- bia\=leo (true) (P = bia, X = tenis)
        |
        |-- joga(leo, tenis) (P = leo)
            |
            |-- leo\=leo (false) (P = leo, X = tenis) (Backtrack)
```

3.2.

Considere o programa a seguir:

O predicado num classifica números em três categorias: positivos, nulo e negativos. Esse predicado, da maneira como está definido, realiza retro□cesso desnecessário. Explique por que isso acontece e, em seguida, utilize cortes para eliminar esse retrocesso.

```
num(N,positivo) :- N>0.
num(0,nulo).
num(N,negativo) :- N<0.
```

R: O retrocesso desnecessário ocorre quando o Prolog, ao realizar uma consulta, encontra múltiplas cláusulas que correspondem a um predicado e, após encontrar a primeira correspondência, continua procurando outras correspondências para esse predicado, mesmo que a primeira seja suficiente para produzir a resposta desejada. Isso pode levar a uma busca mais lenta e ineficiente.

No caso do predicado num/2 que classifica números em três categorias (positivos, nulo e negativos), o retrocesso desnecessário ocorre quando um número é positivo ou negativo. Após encontrar uma correspondência, o Prolog ainda verifica as outras cláusulas para determinar se há outras correspondências, o que não é necessário.

Para eliminar o retrocesso desnecessário, podemos usar cortes (!) para indicar que, após uma correspondência ser encontrada, não devemos considerar outras opções. Aqui está a definição do predicado num/2 com o uso de cortes:

```
num(N,positivo) :- N > 0, !.
num(0,nulo) :- !.
num(N,negativo) :- N < 0.
```

Neste caso, o corte ! é usado após cada cláusula para indicar que, uma vez que uma correspondência é encontrada, não é necessário procurar por outras correspondências para o mesmo predicado. Isso elimina o retrocesso desnecessário e torna a consulta mais

eficiente. Agora, se um número for positivo, nulo ou negativo, o Prolog não buscará alternativas desnecessárias após encontrar uma correspondência adequada.

3.3.

Suponha que o predicado fail não existisse em Prolog. Qual das duas definições a seguir poderia ser corretamente usada para causar falhas? a) falha :- (1=1). b) falha :- (1=2).

R: A opção "a) falha :- (1=1)." não causaria uma falha, pois "1=1" é uma expressão logicamente verdadeira em Prolog, e a unificação seria bem-sucedida. Portanto, a cláusula "falha" não falharia nesse caso. A cláusula "falha :- (1=2)." tenta unificar o resultado da expressão "1=2" com "falha". No entanto, como "1=2" é uma expressão logicamente falsa em Prolog, essa unificação falhará. Isso levará à falha da cláusula "falha", o que é o resultado desejado.

3.4.

Considere o programa a seguir:

```
animal(cão).
animal(canário).
animal(cobra).
animal(morcego).
animal(gaivota).
voa(canário).
voa(morcego).
voa(gaivota).

dif(X,X) :- !, fail.
dif(_,_).

pássaro(X) :- animal(X), voa(X), dif(X,morcego).
```

Desenhe a árvore de busca necessária para responder a consulta ?- pássaro(X).

Em seguida, execute o programa para ver se as respostas do sistema correspondem àquelas que você encontrou.

```
?- pássaro(X)
    |
    |-- animal(cão) (Fail) (Backtrack)
    |
    |-- animal(canário)
    |   |
    |   |-- voa(canário)
    |   |   |
    |   |   |-- dif(canário, morcego)
    |   |   |   |
    |   |   |   |-- fail (Cut)
    |   |   |   |
    |   |   |   |-- dif(canário, morcego) (Exit)
    |   |   |
```

```
|   |   |-- pássaro(canário) (Exit)
|   |
|   |-- animal(cobra) (Fail) (Backtrack)
|   |
|   |-- animal(morcego)
|   |   |
|   |   |-- voa(morcego)
|   |   |   |
|   |   |   |-- dif(morcego, morcego)
|   |   |   |   |
|   |   |   |   |-- fail (Cut)
|   |   |   |   |
|   |   |   |   |-- dif(morcego, morcego) (Fail)
|   |   |   |
|   |   |   |-- pássaro(morcego) (Exit)
|   |   |
|   |   |-- animal(gaivota)
|   |   |   |
|   |   |   |-- voa(gaivota)
|   |   |   |   |
|   |   |   |   |-- dif(gaivota, morcego)
|   |   |   |   |   |
|   |   |   |   |   |-- dif(gaivota, morcego) (Exit)
|   |   |   |   |
|   |   |   |   |-- pássaro(gaivota) (Exit)
```

```
X = canário ;
X = gaivota.
```

4. Capítulo 4: Programação Recursiva

4.1.

Defina um predicado recursivo para calcular o produto de dois números naturais usando apenas soma e subtração.

```
mult(X,0,0):- X >= 0,!.
mult(X,1,X):- X >= 0,!.
mult(X,Y,Z):-
            X >= 0,
                        Y >= 0,
            Y1 is Y - 1,
            mult(X,Y1,Z1),
            Z is X + Z1,!.
```

Explicação: O predicado mult/3 recebe dois números naturais X e Y e retorna o produto Z. Para isso, ele usa que o produto de X e Y é igual a X somado Y vezes. Portanto, o predicado mult/3 usa recursão para calcular o produto de X e Y. Primeiro, ele verifica se X e Y são números naturais (ou seja, se são maiores ou iguais a 0). Se X e Y forem números naturais, ele verifica se Y é igual a 0 ou 1. Se for, o produto é igual a 0 ou X, respectivamente. Caso contrário, ele calcula o produto de X e Y-1 e adiciona X ao resultado. O corte! é usado para evitar o retrocesso desnecessário.

4.2.

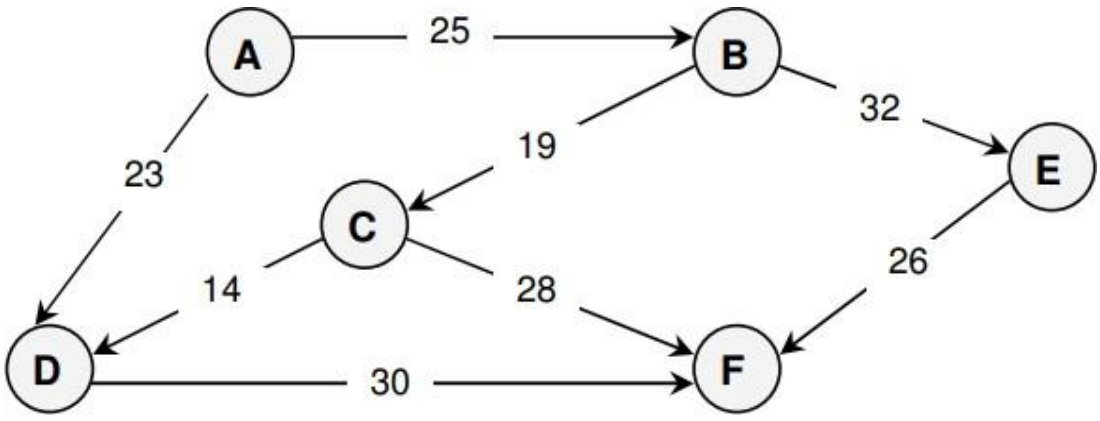
Defina um predicado recursivo exibir um número natural em binário.

```
bin(0, '0').
bin(1, '1').

bin(N, B) :-
    N > 1,
    R is N mod 2,
    Q is N // 2,
    bin(Q, BQ),
    atom_concat(BQ, R, B).
```

4.3.

O grafo a seguir representa um mapa, cujas cidades são representadas por letras e cujas estradas (de sentido único) são representados por números, que indicam sua extensão em km.



a) Usando o predicado `estrada(O,D,KM)` , crie um programa para representar esse mapa. b) Defina a relação transitiva `dist(A,B,D)` , que determina a distância `D` entre duas cidades `A` e `B` .

R: a) O programa que representa o mapa é o seguinte:

```
estrada(a,b,25).
estrada(a,d,23).
estrada(b,c,19).
estrada(b,e,32).
estrada(c,f,28).
estrada(c,d,14).
estrada(d,f,30).
estrada(e,f,26).
```

b) A relação transitiva dist/3 é definida da seguinte forma:

```
dist(A,B,D) :- estrada(A,B,D).
dist(A,B,D) :- estrada(A,C,D1), dist(C,B,D2), D is D1 + D2.
```

teste:

```
?- dist(a,b,D).
D = 25 ;
false.

?- dist(a,f,D).
D = 53 ;
false.
```