Václav Račanský

Umělá inteligence

Zápisy z přednášky zpracoval:
Jan Šerák

23. května 1995

Obsah

Seznam obrázků

1 Opakování jazyka Prolog

1.1 Fibonacciho čísla

Zde uvádíme dvě varianty, jak se na problém Fibonacciho čísel dívat. První z nich je klasický přístup logického programování, t.j. přístup formulování problému.

```
f(X,X) :- X=<1.
f(X,Y) :- X1 is X-1, X2 is X-2, f(X1,Y1), f(X2,Y2), Y is Y1+Y2.
```

Tento program počítá Fibonacciho čísla, ovšem složitost, s jakou pracuje, je exponenciální. Nyní si uveďme týž program, s jistým fíglem, který sníží složitost našeho programu na složitost lineární.

```
f(X,X) :- X=<1.
f(X,Y) :- X1 is X-1, X2 is X-2, f(X1,Y1), f(X2,Y2), Y is Y1+Y2,
          asserta(f(X,Y)).
```

1.2 Třídící algoritmy

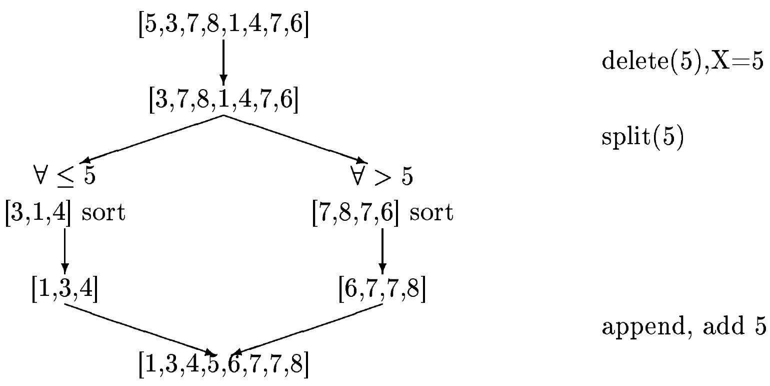
1.2.1 Bublinkové třídění

První ze třídících algoritmů, které si uvedeme je bublinkové třídění. Není na něm vůbec nic náročného:

```
bubblesort(List,Sorted) :- swap(List,List1),!,bubblesort(List1,Sorted).
bubblesort(Sorted,Sorted).
swap([X,Y|Rest],[Y,X|Rest]) :- X>Y.
swap([Z|Rest],[Z|Rest1]) :- swap(Rest,Rest1).
```

1.2.2 Quicksort

Princip algoritmu `Quicksort` ukazuje obrázek 1. Tedy zapsáno v Prologu:



Obrázek 1: Princip algoritmu `Quicksort`

```
quicksort([],[]).
quicksort([X|Tail],Sorted) :- split(X,Tail,Small,Big),quicksort(Small,SortedSmall),
                              quicksort(Big,SortedBig),
                              append(SortedSmall,[X|SortedBig]).
split(X,[],[],[]).
split(X,[Y|Tail],[Y|Small],Big) :- X>Y,!,split(X,Tail,Small,Big).
split(X,[Y|Tail],Small,[Y|Big]) :- split(X,Tail,Small,Big).
```

1.3 Práce se seznamy

1.3.1 Smazání prvku ze seznamu

Smazání prvku ze seznamu se provede tak, pokud vymazávaný prvek `X` není v hlavě seznamu, hlava se odstraní a pokračuje se na zbytku seznamu. Je-li prvek `X` v hlavě seznamu, odstraní se a program končí úspěchem.

```
del(X,[X|Tail],Tail).
del(X,[Y|Tail],[Y|Tail1]) :- del(X,Tail,Tail1).
```

1.3.2 Vložení prvku do seznamu

Pro vkládání prvku do seznamu si uvedeme dva predikáty. Predikát `insert` je konstruován jako logický důsledek existence predikátu `del` a vztahu mezi operacemi vkládání a vybírání do/ze seznamu.

```
insert(X,List,List1) :- del(X,List1,List).
```

Predikát `insert1` je použitelný v praxi.

```
insert1(X,List,[X|List]).
```

1.3.3 Permutace

Predikáty `perm1` a `perm2` generují permutace `P` z prvků seznamu `L` pomocí predikátů `insert` a `del`.

```
perm1([],[]).
perm1([X|L],P) :- perm1(L,L1),insert(X,L1,P).
perm2([],[]).
perm2(L,[X|P]) :- del(X,L,L1),perm2(L1,P).
```

1.4 Problém osmi dam

V tomto odstavci se podíváme na problém osmi dam. Tento problém lze formulovat například takto. Rozestavějte po šachovnici 8 dam tak, aby se žádné dvě vzájemně neohrožovaly.

Pro řešení tohoto problému si vybereme jako datovou strukturu osmiprvkový seznam, reprezentující osm dam. Každý prvek seznamu má tvar `A/B`, kde `A` je horizontální a `B` vertikální souřadnice polohy dámy na šachovnici. Např. situaci na obr. 2 zobrazíme seznamem:

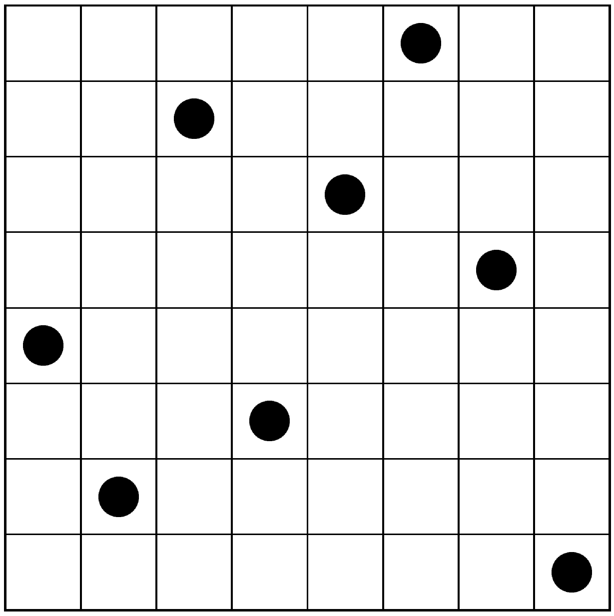
```
[1/4,2/2,3/7,4/3,5/6,6/8,7/5,8/1]
```

1.4.1 Řešení č. 1

Toto řešení je konstruováno „od boku". Predikát `template` vygeneruje naši datovou strukturu, v níž je splněno, že v každém sloupci stojí právě jedna dáma. Predikát `solution` potom generuje všechna možná rozestavění dam, kdy je každá ve svém sloupci sama, a pomocí predikátu `noatack` kontroluje, zda se některé dvě dámy neohrožují.

```
solution([]).
solution([X/Y|Others]) :- solution(Others), member(Y,[1,2,3,4,5,6,7,8]),
                          noattack(X/Y,Others).
noattack(_,[]).
noattack(X/Y,[X1/Y1|Others]) :- Y=\=Y1, Y1-Y=\=X1-X, Y1-Y=\=X-X1,
                                noattack(X/Y,Others).
template([1/Y1,2/Y2,3/Y3,4/Y4,5/Y5,6/Y6,7/Y7,8/Y8]).
```

Toto řešení problému negeneruje všechny možné permutace řádků, tedy všechny možné pozice všech dam v rámci jejich sloupců. Predikát `solution` umístí vždy i-tou dámu (pro $i = 1, \ldots, 8$) tak, aby neohrožovala žádnou z dosud rozmístěných.



Obrázek 2: Příklad rozestavění dam na šachovnici

1.4.2 Řešení č. 2

Datovou strukturou použitou v tomto řešení je seznam osmi vertikálních souřadnic, protože se předpokládá, že každá z dam leží ve svém vlastním sloupci, t.j. `[Y1,Y2,Y3,Y4,Y5,Y6,Y7,Y8]`.

```
solution(Queens) :- perm([1,2,3,4,5,6,7,8],Queens),safe(Queens).
safe([]).
safe([Queen|Others]) :- safe(Others),noattack(Queen,Others,1).
noattack(_,[],_).
noattack(Y,[Y1|YList],Xdist) :- Y1-Y=\=Xdist, Y-Y1=\=Xdist, Dist1 is Xdist+1,
                                noattack(Y,YList,Dist1).
```

Toto řešení je horší, neboť jsou generovány všechny možné polohy dam ve svých sloupcích. Predikát `perm` vždy vygeneruje novou permutaci řádků a predikát `safe` vyzkouší, zda je to přípustné rozestavění.

1.4.3 Řešení č. 3

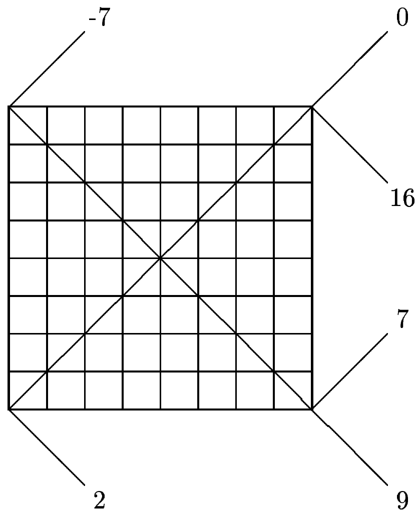
V tomto řešení použijeme nikoli souřadnice x a y (sloupec,řádek), ale tzv. souřadnice diagonály u a v. Tedy na šachovnici aplikujeme transformaci souřadnic:

$$\begin{aligned} u &= x - v \\ v &= x + v \end{aligned}$$

Tedy intervaly $D_x = [1..8]$ a $D_y = [1..8]$ přejdou do $D_u = [-7..7]$ a $D_v = [2..16]$. Názorně to ukazuje obrázek 3.

Text programu bude tedy vypadat takto:

```
solution(YList) :- sol(YList,[1,2,3,4,5,6,7,8],[1,2,3,4,5,6,7,8],
                          [-7,-6,-5,-4,-3,-2,-1,0,1,2,3,4,5,6,7],
                          [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).
sol([],[],Dy,Du,Dv).
sol([Y|YList],[X|Dx1],Dy,Du,Dv) :- del(Y,Dy,Dy1),U is X-Y,del(U,Du,Du1),V is X+Y,
                                   del(V,Dv,Dv1), sol(YList,Dx1,Dy1,Du1,Dv1).
```



Obrázek 3: Transformace souřadnic na šachovnici

Tento algoritmus je z uvedených algoritmů nejlepší. Nutno ovšem poznamenat, že predikát `del` musí skončit neúspěchem (`fail`), pokud není nalezen hledaný prvek.

2 Grafy

2.1 Binární strom

V této kapitole si uvedeme několik algoritmů, provádějících základní operace nad binárními stromy.

2.1.1 Přidávání do binárního stromu

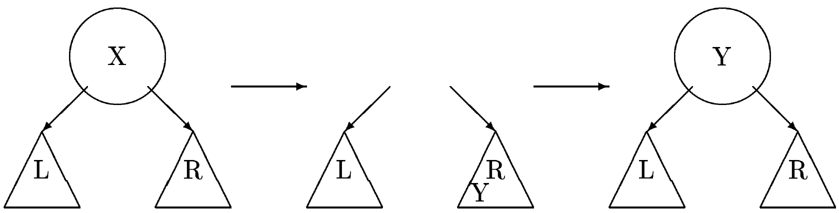
```
addleaf(nil,X,t(nil,X,nil)).
addleaf(t(Left,X,Right),X,t(Left,X,Right)).
addleaf(t(Left,Root,Right),X,t(Left1,Root,Right)) :- Root>X,addleaf(Left,X,Left1).
addleaf(t(Left,Root,Right),X,t(Left,Root,Right1)) :- Root<X,addleaf(Right,X,Right1).
```

Analýzu programu necháme na laskavém čtenáři. Nutno však podotknout, že predikát na odstranění prvku z binárního stromu nelze definovat:

```
del(D,X,D1) :- addleaf(D1,X,D).
```

2.1.2 Odebírání z binárního stromu

Princip „přestavby" binárního stromu při odstraňování kořene ukazuje obrázek 4.



Obrázek 4: Princip odstraňování prvku z binárního stromu

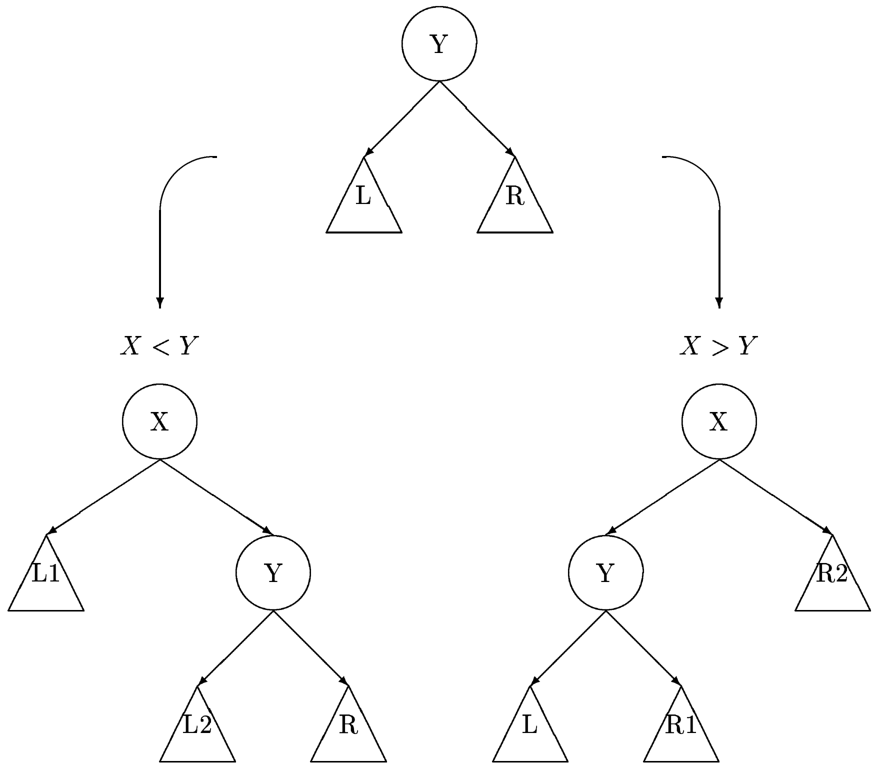
Zbývá tedy ukázat zdrojový text.

```
delleaf(t(nil,X,Right),X,Right).
delleaf(t(Left,X,nil),X,Left).
delleaf(t(Left,X,Right),X,t(Left,Y,Right1)) :- delmin(Right,Y,Right1).
delleaf(t(Left,Root,Right),X,t(Left1,Root,Right)) :- X<Root,delleaf(Left,X,Left1).
delleaf(t(Left,Root,Right),X,t(Left,Root,Right1)) :- X>Root,delleaf(Right,X,Right1).
delmin(t(nil,Y,R),Y,R).
delmin(t(Left,Root,Right),Y,t(Left1,Root,Right)) :- delmin(Left,Y,Left1).
```

2.1.3 Vkládání/odebírání do/z binárního stromu

Princip algoritmu, který zde uvádíme, ukazuje obrázek 5.

```
add(D,X,D1) :- addroot(D,X,D1).
add(t(L,Y,R),X,t(L1,Y,R)) :- gt(Y,X),add(L,X,L1).
add(t(L,Y,R),X,t(L,Y,R1)) :- gt(X,Y),add(R,X,R1).
addroot(nil,X,t(nil,X,nil)).
addroot(t(L,X,R),X,t(L,X,R)).
addroot(t(L,Y,R),X,t(L1,X,t(L2,Y,R))) :- gt(Y,X),addroot(L,X,t(L1,X,L2)).
addroot(t(L,Y,R),X,t(t(L,Y,R1),X,R2)) :- gt(X,Y),addroot(R,X,t(R1,X,R2)).
```



Obrázek 5: Princip vkládání do binárního stromu

Definici predikátu `gt` ponecháváme na konečném uživateli. Uvedený program předpokládá, že `gt(X,Y)` uspěje, pokud je vrchol `X` „větší" než vrchol `Y`.

Poznamenejme, že tento program funguje i „obráceně"[1]; jinými slovy, lze bez problémů zapsat program pro odstraňování prvku z binárního stromu:

```
del(D,X,D1) :- add(D1,X,D).
```

2.1.4 Tisk binárního stromu

V tomto odstavci si uvedeme program na tisk stromu. Bude fungovat tak, že převede datovou strukturu stromu např.

```
t(
  t(
    t(nil,1,nil
    ),3,
    t(nil,4,nil
    )
  ),5,
  t(
    t(nil,6,
      t(nil,7,nil
      )
    ),8,
    t(nil,9,nil
    )
  )
)
```

do tvaru, který je uveden na obr. 6.

```
--------------
        9
    8
            7
        6
5
        4
    3
        1
--------------
```

Obrázek 6: Příklad výstupu programu `show`

[1] Salonšampon a kondicionér v jednom! Two in one!

Program tedy bude vypadat takto:

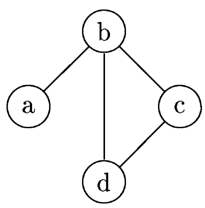
```
show(T) :- show2(T,0).
show2(nil,_).
show2(t(L,X,R),Indent) :- Ind2 is Indent+2,show2(R,Ind2),tab(Indent),
                          write(X),nl,show2(L,Ind2).
```

2.2 Reprezentace grafů

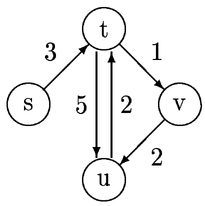
Některé způsoby reprezentace grafů v Prologu:

1. `graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])`. Tato reprezentace znázorňuje neorientovaný graf jako predikát `graph(V,E)`, kde `V` je seznam vrcholů grafu a `E` je seznam hran grafu. Každá hrana je tvaru `e(V1,V2)`, kde `V1` a `V2` jsou vrcholy grafu; viz obr. 7.
2. `digraph([s,t,u,v],[a(s,t,3),a(t,v,1),a(t,u,5),a(u,t,2),a(v,u,2)])` znázorňuje orientovaný graf také jako usp. dvojici seznamů vrcholů a hran, které jsou tvaru `a(PočátečníV,KoncovýV,CenaHrany)`; viz obr. 8.
3. Takový orientovaný graf je uložen v programové databázi jako posloupnost faktů a jednoho pravidla:

```
e(g3,a,b).
e(g3,b,c).
e(g3,b,d).
e(g3,c,d).
e(X,A,B) :- e(X,B,A).
```



Obrázek 7: Příklad neorientovaného grafu



Obrázek 8: Příklad orientovaného grafu

2.3 Cesty v grafech

Zde si uvedeme několik algoritmů pro vyhledávání cest v grafech. U každého bude uvedeno, který druh reprezentace grafu používá.

Jako první si uvedeme program, který najde nějakou cestu v neorientovaném grafu. Program lze spustit dotazem `?- path(A,Z,G,P)`. Program `path` v grafu `G` najde z vrcholu `A` do vrcholu `Z` cestu `P`. Program předpokládá, že graf `G` bude reprezentován ve tvaru 1.

```
path(A,Z,Graph,Path) :- path1(A,[Z],Graph,Path).
path1(A,[A|Path1],_,[A|Path1].
path1(A,[Y|Path1],Graph,Path) :- adjacent(X,Y,Graph),not member(X,Path1),
                                 path1(A,[X,Y|Path1],Graph,Path).
adjacent(X,Y,graph(Nodes,Edges)) :- member(e(X,Y),Edges);member(e(Y,X),Edges).
```

Druhý algoritmus je obdoba předešlého. Hledáme libovolnou cestu z jednoho vrcholu do druhého a její cenu v ohodnoceném neorientovaném grafu. Požadovaná reprezentace grafu plyne z tvaru predikátu `adjacent`. Po úpravě tohoto predikátu lze program transformovat na libovolnou reprezentaci ohodnoceného (ne)orientovaného grafu.

```
path(A,Z,Graph,Path,Cost) :- path1(A,[Z],0,Graph,Path,Cost).
path1(A,[A|Path1],Cost1,Graph,[A|Path1],Cost1).
path1(A,[Y|Path1],Cost1,Graph,Path,Cost) :- adjacent(X,Y,CostXY,Graph),
     not member(X,Path1),Cost2 is Cost1+CostXY,
     path1(A,[X,Y|Path1],Cost2,Graph,Path,Cost).
adjacent(X,Y,CostXY,Graph) :- member(X-Y/CostXY,Graph);member(Y-X/CostXY,Graph).
```

2.4 Kostra grafu

Dalším důležitým grafovým algoritmem je konstrukce kostry grafu. Proto si jej zde také uvedeme.

```
stree(Graph,Tree) :- member(Edge,Graph),spread([Edge],Tree,Graph).
spread(Tree1,Tree,Graph) :- addedge(Tree1,Tree2,Graph),spread(Tree2,Tree,Graph).
spread(Tree,Tree,Graph) :- not addedge(Tree,_,Graph).
addedge(Tree,[A-B|Tree],Graph) :- adjacent(A,B,Graph),node(A,Tree),
                                  not node(B,Tree).
adjacent(A,B,Graph) :- member(A-B,Graph);member(B-A,Graph).
node(A,Graph) :- adjacent(A,_,Graph).
```

3 Prohledávání stavového prostoru

Prohledávání stavového prostoru je jednou z nejzákladnějších metod Umělé inteligence. S tímto přístupem už jsme se částečně seznámili u problému osmi dam. Takže víme o co jde, tedy s chutí do toho.

3.1 Prohledávání stavového stromu

Základem všech prohledávacích algoritmů je tato kostra, která hledá vrchol `N` a v případě úspěchu vrátí i cestu, která ke hledanému vrcholu vede.

```
solve(N,[N]) :- goal(N).
solve(N,[N|Sol1]) :- s(N,N1),solve(N1,Sol1).
```

Predikát `goal(N)` uspěje, pokud `N` je hledané řešení. Ve většině dalších programů budeme definici tohoto predikátu ponechávat na konečném uživateli.

Predikát `s(m,n)` uspěje, pokud (m, n) je hrana stavového stromu.

3.2 Prohledávání do hloubky

Nejprve si v tomto odstavci uvedeme nejjednodušší verzi prohledávání do hloubky[2].

```
solve(Node,Solution) :- depth_first_search([],Node,Solution).
depth_first_search(Path,Node,[Node|Path]) :- goal(Node).
depth_first_search(Path,Node,Sol) :- s(Node,Node1),
     not member(Node1,Path),depth_first_search([Node|Path],Node1,Sol).
```

Výše uvedený program je korektní jen zdánlivě. Lze jej použít pouze na prohledávání do hloubky pouze u konečných grafů. Jednoduchým přidáním „zarážky" v predikátu `depth_first_search` jej však lze transformovat na korektní program.

Konstruujme tedy predikát `depth_first_search2`, který pokud neuspěje do určité dosažené hloubky, pak skončí neúspěchem.

```
depth_first_search2(Node,[Node],_) :- goal(Node).
depth_first_search2(Node,[Node|Sol],MaxDepth) :- MaxDepth>0,s(Node,Node1),
     Max1 is MaxDepth-1,depth_first_search2(Node1,Sol,Max1).
```

3.3 Prohledávání do šířky

Během prohledávání do šířky[3] si musí program uchovávat všechny rozpracované cesty z kořene do každého vrcholu v dané úrovni od kořene.

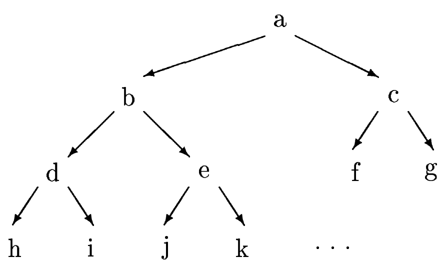
Tedy postup při prohledávání stromu z obr. 9 bude tedy vypadat takto:

1. `[[a]]`
2. `[[b,a],[c,a]]`
3. `[[c,a],[d,b,a],[e,b,a]]`
4. `[[d,b,a],[e,b,a],[f,c,a],[g,c,a]]`
5. `[[h,d,b,a],[i,d,b,a],[j,e,b,a],...]`

Než začneme psát algoritmus prohledávání do šířky, zopakujme si ještě činnost vestavěného prologovského predikátu `bagof(X,P,L)`. Tento predikát postupně vyhodnocuje `P` a všechny vyhovující instance `X` řadí do seznamu `L`.

[2] Obvykle je tento algoritmus uváděn pod názvem „Depth First Search"

[3] Prohledávání do šířky se obvykle nazývá „Breadth First Search"



Obrázek 9: Strom prohledávání do šířky

```
solve(Start,Solution) :- breadth_first_search([[Start]],Solution).
breadth_first_search([[Node|Path]|_],[Node|Path]) :- goal(Node).
breadth_first_search([[N|Path]|Paths],Solution) :-
     bagof([M,N|Path],(s(N,M),not member(M,[N|Path])),NewPaths),
     not NewPaths=[],append(Paths,Newpaths,Path1),!,
     breadth_first_search(Path1,Solution);
     breadth_first_search(Paths,Solution).
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

Poznamenejme ještě, že operátor „`,`" má prioritu před operátorem „`;`", t. j. implicitní závorkování predikátu

```
p :- a,b;c.
```

je

```
p :- (a,b);c.
```

Predikát `append` tak, jak je zde definován však svojí složitostí způsobuje velkou složitost celého algoritmu. Proto si zde uvedeme řešení, které je založeno na rozdílových seznamech. Princip si uveďme na příkladu predikátu `concat`, který v jednom kroku spojí dva řetězce do jednoho:

```
concat(A1-Z1,Z1-Z2,A1-Z2).
```

Dotazem `concat([a,b,c|T1]-T1,[d,e|T2]-T2,List-[]).` pak obdržíme rovnou v proměnné `List` hledaný seznam.

```
solve(Start,Solution) :- bfs([[Start]|Z]-Z,Solution).
bfs([[Node|Path]|_]-_,[Node|Path]) :- goal(Node).
bfs([[N|Path]|Paths]-Z,Solution) :-
     bagof([M,N|Path],(s(N,M),not member(M,[N|Path])),New),
     append(New,ZZ,Z),!,bfs(Paths-ZZ,Solution);
     Paths=\=Z,bfs(Paths-Z,Solution).
```

Algoritmus prohledávání do šířky, jak jsme jej zde uvedli, má veliké nároky na paměť[4]. Proto si zde uveďme ještě jedno řešení Breadth First Search, které paměť šetří.

V předešlých řešeních jsme rozpracované cesty ukládali v seznamu všech rozpracovaných cest. Tím docházelo k tomu, že již ve 3. úrovni prohledávání byl kořen stromu uložen v tomto seznamu čtyřikrát. V následujícím řešení tuto redundanci odstraníme tím, že zavedeme pro každý vrchol, kterým prohledávání prošlo, strukturu:

- buď `l(N)`, pokud je `N` v současné době list;
- nebo `t(N,Subs)`, kde `Subs` je seznam synů vrcholu `N`.

[4]Neboť musí udržovat všechny rozpracované cesty

Takže se nám např. seznam

```
[[d,b,a],[e,b,a],[f,c,a],[g,c,a]]
```

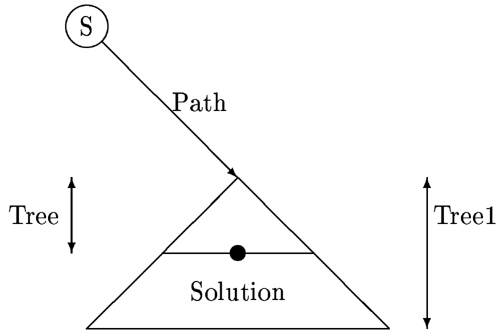
„smrští" do struktury

```
t(a,[t(b,[l(d),l(e)]),t(c,[l(f),l(g)])])
```

Základem následujícího programu je predikát `expand`. Funkci predikátu

```
expand(Path,Tree,Tree1,Solved,Solution).
```

demonstruje obr. 10.



Obrázek 10: Princip chování predikátu `expand`

A nyní si uveďme vlastní program:

```
solve(Start,Solution) :- breadth_first_search(l(Start),Solution).
breadth_first_search(Tree,Solution) :- expand([],Tree,Tree1,Solved,Solution),
     (Solved=yes;Solved=no,breadth_first_search(Tree1,Solution)).
expand(P,l(N),_,yes,[N|P]) :- goal(N).
expand(P,l(N),t(N,Subs),no,_) :- bagof(l(M),(s(N,M),not member(M,P)),Subs).
expand(P,t(N,Subs),t(N,Subs1),Solved,Sol) :- expandall([N|P],Subs,[],Subs1,Solved,Sol).
expandall(_,[],[T|Ts],[T|Ts],no,_).
expandall(P,[T|Ts],Ts1,Subs1,Solved,Sol) :- expand(P,T,T1,Solved1,Sol),
     (Solved1=yes,Solved=yes;Solved1=no,!,expandall(P,Ts,[T1|Ts1],Subs1,Solved,Sol));
     expandall(P,Ts,Ts1,Subs1,Solved,Sol).
```

3.4 Nalezení nejlepší cesty

V tomto odstavci se budeme zabývat problémem nalezení nejlepší cesty[5] ve stromu vzhledem k nějakému ohodnocení hran c tohoto stromu. Tento algoritmus, jak uvidíme dále, bude pro nás základem pro tzv. *prohledávání stavového prostoru*.

Pro rozpracované cesty, resp. pro navštívené uzly, budeme uchovávat cenu, kterou jsme museli zaplatit, než jsme do něj došli – g. Dále u každého uzlu budeme uchovávat odhad ceny, kterou zaplatíme při přechodu do následujícího uzlu – h. V dalším postupu se pak budeme v uzlu n rozhodovat podle nejmenšího $f(n) = g(n) + h(n)$.

1. Budeme uchovávat strukturu `l(N,F/G)` pro uzel n, který je momentálně list a kde `G`$=g(n)$ a `F`$=f(n)$.

2. Budeme uchovávat strukturu `t(N,F/G,Subs)` pro uzel n, který je vnitřním uzlem stromu prohledávání. `Subs` jsou (neprázdné) podstromy, uspořádány podle f-hodnot. `G` má týž význam jako v předchozím bodě a `F` je „inovovaná" f-hodnota uzlu n, t.j. f-hodnota nejnadějnějšího[6] následníka uzlu n.

[5]Nalezení nejlepší cesty je obvykle označnováno jako Best Search

[6]Exaktně řečeno: je-li T strom s kořenem n, jehož následníky jsou $m_1, m_2, \ldots$, pak $f(T) = \min_i(f(m_i))$.

A nyní vlastní program:

```
bestsearch(Start,Solution) :- biggest(Big), expand([],l(Start,0/0),Big,_,yes,Solution).
expand(P,l(N,_),_,_,yes,[N|P]) :- goal(N).
expand(P,l(N,F/G),Bound,Tree1,Solved,Sol) :- F=<Bound,
     (bagof(M/C,(s(N,M,C),not member(M,P)),Succ),!,succlist(G,Succ,Ts),bestf(Ts,F1),
          expand(P,t(N,F1/G,Ts),Bound,Tree1,Solved,Sol);Solved=never).
expand(P,t(N,F/G,[T|Ts]),Bound,Tree1,Solved,Sol) :- F=<Bound, bestf(Ts,BF),
     min(Bound,BF,Bound1),expand([N|P],T,Bound1,T1,Solved1,Sol),
     continue(P,t(N,F/G,[T1|Ts]),Bound,Tree1,Solved1,Solved,Sol).
expand(_,t(_,_,[]),_,_,never,_) :- !.
expand(_,Tree,Bound,Tree,no,_) :- f(Tree,F), F>Bound.
continue(_,_,_,_,yes,yes,Sol).
continue(P,t(N,F/G,[T1|Ts]),Bound,Tree1,Solved1,Solved,Sol) :-
     (Solved=no,insert(T1,Ts,NTs);Solved=never,NTs=Ts),
     bestf(NTs,F1),expand(P,t(N,F1/G,NTs),Bound,Tree1,Solved,Sol).
succlist(_,[],[]).
succlist(G0,[N/C|NCs],Ts) :- G is G0+C,h(N,H),F is G+H,succlist(G0,NCs,Ts1),
     insert(l(N,F/G),Ts1,Ts).
insert(T,Ts,[T|Ts]) :- f(T,F),bestf(Ts,F1),F=<F1,!.
insert(T,[T1|Ts],[T1|Ts1]) :- insert(T,Ts,Ts1).
f(l(_,F/_),F).
f(t(_,F/_,_),F).
bestf([T|_],F) :- f(T,F).
bestf([],Big) :- biggest(Big).
min(X,Y,X) :- X=<Y,!.
min(X,Y,Y).
```

Predikát `biggest(Big)` nainstanciuje proměnnou `Big` na hodnotu předpokládané horní závory pro cenu nejlepší cesty.

Pro zjednodušení četby algoritmu ještě uveďme význam jednotlivých parametrů predikátu `expand`.

→ `P` — cesta mezi kořenem a `T`

→ `T` — prohledávaný podstrom

→ `B` — f-limita (hranice) pro expandování `T`

← `T1` — `T` expandované v závislosti na `B` (t.j. první, kde f-hodnota „přelezla" `B`)

← `Solved` — `yes`, `no`, `never`

← `Solution` — cesta z kořene do cílového uzlu

Poznamenejme ještě, že predikát

- `s(N,M,C)` udává cenu c, jíž je ohodnocena hrana (n, m).
- `h(N,H)` je tzv. heuristický odhad cesty.

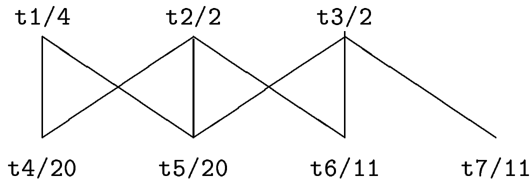
3.5 Rozvrh práce procesorů

Nyní si zde uvedeme praktický příklad prohledávání stavového prostoru. Mějme úlohy `t1,..., t7`, jejichž potřebný čas na zpracování a precedenci úloh ukazuje obr. 11.

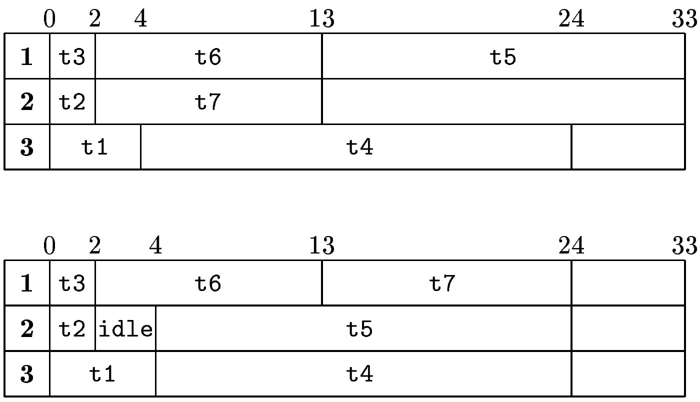
Naším úkolem je rozvrhnout zpracování těchto úloh na třech procesorech, tak aby nebyla porušena precedence úloh a při tom potřebný celkový čas byl minimální. Dvě možná řešení ukazuje obr. 12.

Povšimněme si, že pro nás může někdy být výhodné, když jeden (nebo více) procesor(ů) chvíli počká (idle).

Nyní si tedy formalizujme a implementujme stavy a přechody mezi těmito stavy:



Obrázek 11: Precedence úloh



Obrázek 12: Rozvržení práce procesorů

`nezařazené_úlohy*zařazené_úlohy*čas_ukončení`

např.

`[Task1/D1,Task2/D2,...]*[Task1/F1,Task2/F2,...]*FinTime`

kde `D1, D2,...` jsou doby, potřebné ke zpracování příslušné nezařazené úlohy, a `F1, F2,...` časy ukončení zařazených procesů.

Přechody mezi stavy si definujme takto:

```
s(Tasks1*[_/F|Active]*Fin1,Tasks2*Active2*Fin2,Cost) :- del(Task/D,Tasks1,Tasks2),
     not (member(T/_,Tasks2),before(T,Task)),
     not (member(T1/F1,Active1),F<F1,before(T1,Task)),
     Time is F+D, insert(Task/Time,Active1,Active2,Fin1,Fin2),Cost is Fin2-Fin1.
s(Tasks*[_/F|Active1]*Fin,Tasks*Active2*Fin,0) :- insertidle(F,Active1,Active2).
before(T1,T2) :- prec(T1,T2).
before(T1,T2) :- prec(T,T2),before(T1,T).
insert(S/A,[T/B|L],[S/A,T/B|L],F,F) :- A=<B,!.
insert(S/A,[T/B|L],[T/B|L1],F1,F2) :- insert(S/A,L,L1,F1,F2).
insert(S/A,[],[S/A],_,A).
insertidle(A,[T/B|L],[idle/B,T/B|L]) :- A<B,!.
insertidle(A,[T/B|L],[T/B|L1]) :- insertidle(A,L,L1).
del(A,[A|L],L).
del(A,[B|L],[B|L1]) :- del(A,L,L1).
goal([]*_*_).
```

Na tuto specifikaci přechodů mezi stavy lze pak použít algoritmus nalezení nejlepší cesty „Best Search". Stav, z nějž budeme startovat prohledávání, bude vypadat takto:

```
start([t1/4,t2/2,t3/2,t4/20,t5/20,t6/11,t7/11]*[idle/0,idle/0,idle/0]*0).
```

Nyní si ještě uveďme heuristiku, která nás v tomto řešení „povede" správným směrem. Uvažme, že optimální (nedosažitelný) čas je:

$$Final = \frac{\sum_i (D_i) + \sum_j (F_j)}{m}$$

kde m je počet procesorů. Dále skutečný čas výpočtu všech úloh je:

$$Fin = \max_j (F_j)$$

Potom naše heuristická funkce h bude vypadat takto:

$$H = \begin{cases} Finall - Fin, & Finall > Fin \\ 0, & Final \leq Fin \end{cases}$$

Příslušný predikát `h`, který bude tuto heuristickou funcki počítat, bude vypadat takto:

```
h(Tasks*Processors*Fin,H) :- totaltime(Tasks,TotTime),sumnum(Processors,Ftime,N),
     Finall is (TotTime+Ftime)/N,(Finall>Fin,!,H is Finall-Fin;H=0).
totaltime([],0).
totaltime([_/D|Tasks],T) :- totaltime(Tasks,T1),T is T1+D.
sumnum([],0,0).
sumnum([_/T|Procs],FT,N) :- sumnum(Procs,FT1,N1),N is N1+1,FT is FT1+T.
prec(t1,t4).
prec(t1,t5).
...
```

3.6 Puclíček

„Puclíček" je hra, podobná „Patnáctce"[7]. Je jednodušší v tom smyslu, že se nehraje s patnácti kameny na ploše 4×4, ale s osmi kameny na ploše 3×3 polí. Cílem hry je posunovat kameny tak, až se dojde do situace, která je na obr. 13.

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

Obrázek 13: Cílová situace hry Puclíček

My budeme kódovat konfigurace Puclíčka do seznamu uspořádaných dvojic `X/Y`, kde `X` je číslo sloupce a `Y` číslo řádku. Na prvním místě seznamu bude poloha „díry" (t.j. neobsazeného pole) a pak postupně kamene č. 1, kamene č. 2,..., kamene č. 8. Cílová situace na herní ploše tedy pro nás bude:

```
goal([2/2,1/3,2/3,3/3,3/2,3/1,2/1,1/1,1/2]).
```

Nyní si uveďme predikát, který (stejně jako v předchozím odstavci) definuje přechody ze stavu do stavu:

```
s([Empty|L],[T|L1],1) :- swap(Empty,T,L,L1).
swap(E,T,[T|L],[E|L]) :- d(E,T,1).
swap(E,T,[T1|L],[T1|L1]) :- swap(E,T,L,L1).
d(X/Y,X1/Y1,D) :- dif(X,X1,Dx),dif(Y,Y1,Dy),D is Dx+Dy.
dif(A,B,D) :- D is A-B,D>=0,!;D is B-A.
```

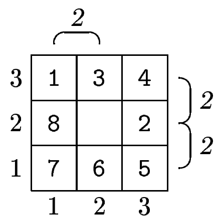
[7] Název „Puclíček" si vymyslel pro tuto hru autor těchto zápisků. Vychází z anglického pojmenování hry v patnáct „Puzzle".

Abychom mohli opět využít algoritmu „Best Search", musíme ještě nadefinovat predikát `h`, který bude určovat heuristickou funkci. Pro tento příklad je vhodná hodnota heuristické funkce ve stavu hry jako součet sumy vzdáleností všech kamenů od správné pozice a trojnásobku „pokut". Tedy:

$$H = \Sigma + 3\Xi$$

kde Ξ je součet pokut. Pokuty se rozdávají za:

- pozici uprostřed 1 trestný bod,
- porušení pořadí 2 trestné body.

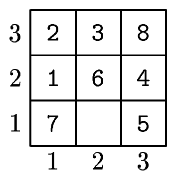


Obrázek 14: Pokuty ve hře Puclíček

V příkladu na obr. 14 je $\Xi = 6$.

```
h([Empty|L],H) :- goal([Empty1|G]),totdist(L,G,D),seq(L,S),H is D+3*S.
totdist([],[],0).
totdist([T|L],[T1|L1],D) :- d(T,T1,D1),totdist(L,L1,D2),D is D1+D2.
seq([First|L],S) :- seq([First|L],First,S).
seq([T1,T2|L],First,S) :- score(T1,T2,S1),seq([T2|L],First,S2),S is S1+S2.
seq([Last],First,S) :- score(Last,First,S).
score(2/2,_,1) :- !.
score(1/3,2/3,0) :- !.
score(2/3,3/3,0) :- !.
...
score(1/2,1/3,0) :- !.
score(_,_,2).
```

Na závěr tohoto odstavce ještě příklad. Algoritmus Best Search dovede s výše uvedenými predikáty `s` a `h` situaci z obr. 15 do vítězného konce v pěti tazích. Ověření necháváme na laskavém čtenáři.



Obrázek 15: Tuto situaci Best Search zvládne v pěti tazích

3.7 AND/OR stromy

Než si začneme povídat o AND/OR stromech, uveďme se do problematiky pomocí notoricky známého příkladu s Hanoiskými věžemi.

3.7.1 Hanoiské věže

Každý ví, co jsou to „Hanoiské věže"[8]. Zkusme se tedy na tento problém podívat analyticky.

Máme tři tyče: A, B a C. Na tyči A je (uspořádaně podle velikosti) nasunuto n kotoučů. Máme za úkol je přeskládat z tyče A pomocí tyče C na tyč B (stručně zapsáno $n(A,B,C)$) bez porušení uspořádání na jednotlivých tyčích. Tento problém se však dá rozložit na tyto fáze:

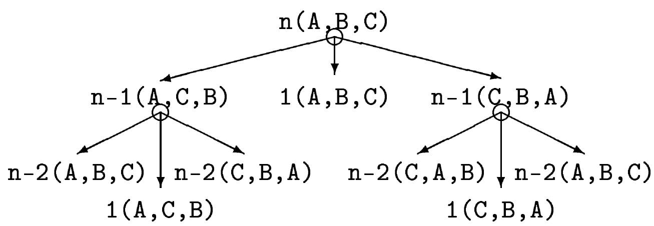
1. přeskládat $n-1$ kotoučů z tyče A pomocí tyče B na tyč C.

2. přeložit 1 kotouč z tyče A na tyč B.

3. přeskládat $n-1$ kotoučů z tyče C pomocí tyče A na tyč B.

Když si to stručně zapíšeme, dostaneme rekurentní vzorec, který problém přeskládání n kotoučů převede na postupně přeskládání $n-1$ kotoučů, přeložení jednoho kotouče (což je v tomto případě pro nás elementární problém) a přeskládání $n-1$ kotoučů.

Když si zakreslíme toto schéma (obr. 16), dostaneme konečný strom, který má v listech pouze elementárí problémy a který pak projdeme postupně z leva do prava a tím získáme řešení celého problému s n kotouči.



Obrázek 16: AND/OR strom hanoiských věží

Nyní si tedy hanoiské věže naprogramujme výše uvedeným způsobem. Poznamenejme ještě, že operátor `to` si musíme nadefinovat.

```
hanoi(1,A,B,C,[A to B]).
hanoi(N,A,B,C,Moves) :- N>1,N1 is N-1,lemma(hanoi(N1,A,C,B,Ms1)),
     hanoi(N1,C,B,A,Ms2),append(Ms1,[A to B|Ms2],Moves).
lemma(P) :- P,asserta((P :- !)).
```

3.7.2 Cesta mezi městy

Nyní si (opět na příkladu) demonstrujme, co je to AND/OR strom. Mějme takový problém. Na obr. 17 je mapka, která nám určuje, mezi kterými městy existují silnice. Města `a,..., e` jsou v Čechách, města `u,..., z` jsou na Moravě a města `k` a `l` jsou hraniční přechody. Máme za úkol najít co nejkratší cestu z města `a` do města `z`.

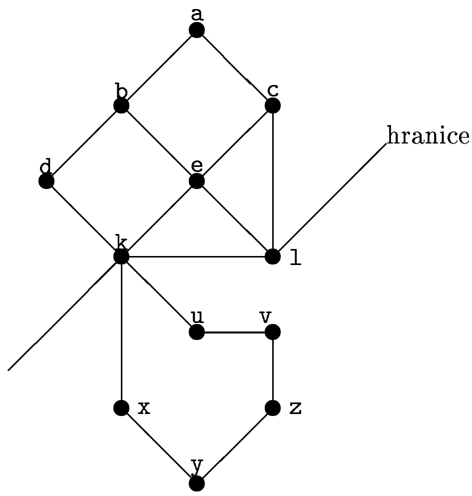
Uvědomme si, že při řešení tohoto problému musíme vzít v potaz klíčové postavení hraničních přechodů `k` a `l`. To značí, že naše úloha se tím rozpadne na nalezení cesty z `a` do některého hraničního přechodu a z tohoto hraničního přechodu do `z`. AND/OR strom tohoto problému je zobrazen na obr. 18. Řešením tohoto problému je podstrom tohoto AND/OR stromu, který nevynechává žádného následníka AND-uzlu.

První řešení, které je nabíledni, je řešení pomocí prologovských prostředků. Pokud reprezentujeme každý OR-uzel `v` s následníky `u1,u2,...,uN` pravidly:

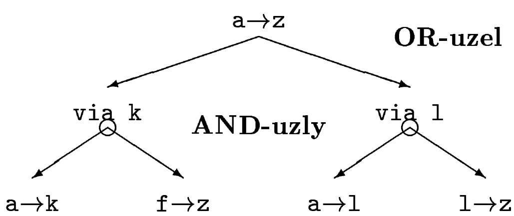
```
v :- u1.
v :- u2.
...
v :- uN.
```

a každý AND-uzel `x` s následníky `y1,y2,...,yM` pravidlem:

[8] A kdo to neví, ať tam běží



Obrázek 17: Mapa měst



Obrázek 18: Příklad AND/OR stromu

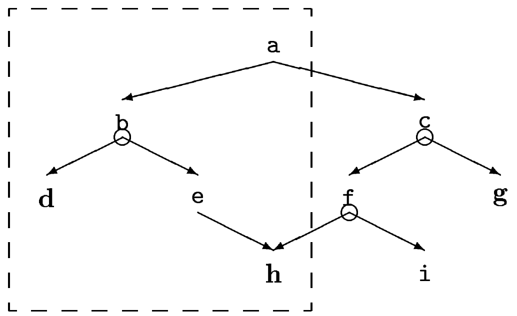
```
x :- y1,y2,...,yM.
```

a pokud uzel `root` je kořenem tohoto AND/OR stromu, pak dotazem:

```
?- root.
```

získáme odpověď na otázku, zda existuje řešení.

Následující zdrojový text reprezentuje AND/OR strom z obr. 19. Poznamenejme ještě, že **tučně** jsou na tomto obrázku označeny cílové uzly a ostatní uzly jsou značeny `strojopisem`.



Obrázek 19: Triviální prohledávání AND/OR stromu.

```
a :- b.
a :- c.
b :- d,e.
e :- h.
c :- f,g.
f :- h,i.
d.
g.
h.
```

Dotazem `?- a.` získáme odpověď `yes`, ale řešení, jako je zobrazeno na obr. 19 v čárkovaném rámečku nám stejně nedá.

3.7.3 AND/OR strom

Ve výše uvedených odstavcích jsme si uvedli příklady AND/OR stromů. Lidsky řečeno je AND/OR strom tedy strom, jehož každý vnitřní uzel má atribut, jehož hodnoty jsou AND a OR. Při prohledávání AND/OR stromu se OR-uzly chovají stejně jako u stromů, ale je nutné při něm projít podstromy všech následníků všech AND-uzlů.

Abychom si lépe mohli AND/OR stromy reprezentovat, zavedeme si tuto relaci:

```
a --> or:[b,c].
b --> and:[d,e].
```

což značí, že z uzlu `a` vedou dvě hrany do uzlů `b,c`, přičemž uzel `a` je OR-uzel. Podobně z `b` vedou dvě hrany do uzlů `d,e`, přičemž uzel `b` je AND-uzel.

Zavedeme tedy operátory:

```
?- op(600,xfx,-->).
?- op(500,xfx,:).
```

Náš AND/OR strom z obr. 19 bude tedy vypadat takto:

```
a --> or:[b,c].
b --> and:[d,e].
c --> and:[f,g].
e --> or:[h].
f --> or:[h,i].
goal(d).
goal(g).
goal(h).
```

3.7.4 Poznámka o prioritách operátorů

Predikátem `op` můžeme definovat operátory. První parametr `op` hlásá prioritu operátoru, třetí parametr jeho identifikátor. Druhý parametr udává typ operátoru podle tabulky 1.

Poloha operátoru	Možnosti definice		
infix	xfx	xfy	yfx
prefix	fx	fy	
postfix	xf	yf	

Tabulka 1: Přehled typů operátorů v Prologu

Výraz Prolog vyhodnocuje takto: je-li argument v závorkách nebo nestrukturovaný, pak má precedenci rovnu 0. Je-li argument struktura, pak má precedenci rovnu precedenci operátoru. Přitom `x` reprezentuje argument, jehož precedence je menší ($<$), než precedence operátoru a `y` reprezentuje argument, jehož precedence je menší nebo rovna ($\leq$) precedenci operátoru.

Uveďme si to na příkladu. Nechť máme operátor pomlčka (-) s precedencí 500 a typu `yfx`. Pak výraz `a-b-c`, v němž se nejprve vyhodnotí `a-b` typu `yfx` vyhovuje, kdežto vyhodnocuje-li se nejprve `b-c`, tomuto typu nevyhovuje.

Dodejme ještě na tomto místě, že prologovský operátor pravidla `:-` je definován s nejvyšší možnou prioritou, a to takto: `?- op(1200,xfx,':-').`

3.8 AND/OR prohledávání do hloubky

Jelikož víme, co je AND/OR strom, a víme, co je prohledávání do hloubky, můžeme rovnou psát algoritmus prohledávání do hloubky pro AND/OR stromy.

```
solve(Node,Node) :- goal(Node).
solve(Node,Node --> Tree) :- Node --> or:Nodes,member(Node1,Nodes),
     solve(Node1,Tree).
solve(Node,Node --> and:Trees) :- Node --> and:Nodes,solveall(Nodes,Trees).
solveall([],[]).
solveall([Node|Nodes],[Tree|Trees]) :- solve(Node,Tree),solveall(Nodes,Trees).
```

3.8.1 AND/OR prohledávání do hloubky s oceněním

Prohledávání do hloubky s oceněním si vyžádá tuto změnu v reprezentace AND/OR stromů:

```
Uzel --> AndOr:[Syn1/Cena1,Syn2/Cena2,...,SynN/CenaN].
```

Mimoto si pro každý uzel N definujeme hodnotu $H(N)$ – odhad obtížnosti stromu řešení:

- $H(N) = h(N)$ pro uzel N momentálně listový;
- $H(N) = 0$ pro elementární problém;
- $H(N) = \min_i (cost(N, N_i) + H(N_i))$ pro OR-uzel N s následníky $N_1, N_2, \ldots$;

- $H(N) = \sum_i (cost(N, N_i) + H(N_i))$ pro AND-uzel N s následníky $N_1, N_2, \ldots$;

Dále si nadefinujeme pro každý uzel N hodnotu $F(N)$:

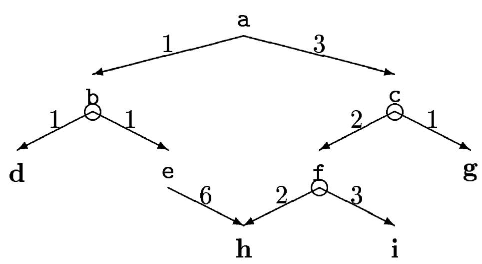
- $F(N) = h(N)$ pro uzel N momentálně listový;
- $F(N) = 0$ pro elementární problém;
- $F(N) = cost(M, N) + \min_i (F(N_i))$ pro OR-uzel N s následníky $N_1, N_2, \ldots$ a bezprostředním předchůdcem M;
- $F(N) = cost(M, N) + \sum_i (F(N_i))$ pro AND-uzel N s následníky $N_1, N_2, \ldots$ a bezprostředním předchůdcem M;

H a F jsou v podstatě tytéž heuristické funkce, ale F bude pro nás výhodnější.

3.8.2 Demonstrace AND/OR prohledávání s oceněním

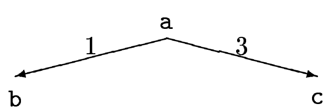
Zde si demonstrujeme na několika obrázcích postup AND/OR prohledávání do hloubky. Budeme si jej demonstrovat na AND/OR stromu z obr. 20.

Poznamenejme ještě, že číslice 1,2,3... uváděné u jednotlivých hran reprezentují cenu hrany a číslice `1,2,3...` pod jednotlivými uzly představují momentální `F`-hodnoty uzlů.



Obrázek 20: Demonstrace AND/OR prohledávání

Expandovat strom z obrázku 20 začínáme z kořene `a` a výsledek expandování je na obr. 21. Rozhodnutí, který z uzlů `b`, `c` budeme dále expandovat, se uskuteční podle minimální aktuální `F`-hodnoty. Jelikož `F(b) < F(c)`, budeme dále expandovat uzel `b`. Výsledek expandování je na obr. 22.

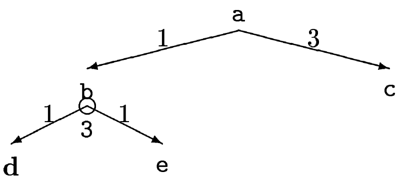


Obrázek 21: `F(b)=1 < F(c)=3`

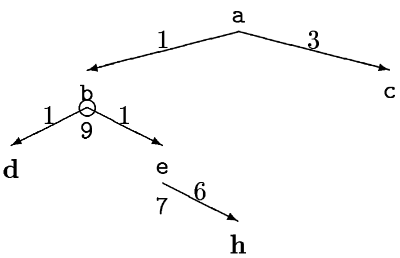
Na tomto obrázku si všimněme, jak se spočítá `F(b)`, je-li `b` AND-uzel. Jelikož v `F`-hodnotách uzlů `b` a `c` nastala nyní rovnost, můžeme pokračovat v expandování uzlu `e`, jak ukazuje obr. 23.

Zde však již `F`-hodnota uzlu `b` převýšila `F`-hodnotu uzlu `c`, takže začneme expandovat uzel `c`, protože se „tváří" jako nadějnější.

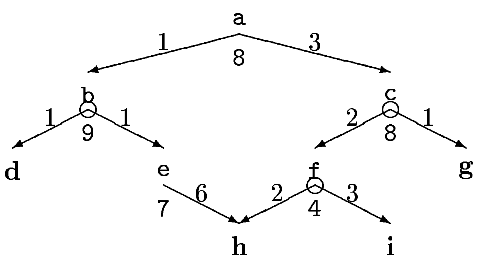
Výsledný prohledaný AND/OR strom ukazuje obr. 24. Konečná `F`-hodnota kořene `a` je také výsledná cena nejlepšího řešení AND/OR stromu.



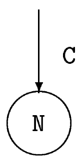
Obrázek 22: F(b) = 3 = F(c)



Obrázek 23: F(b)=9 > 3=F(c)



Obrázek 24: AND/OR strom po AND/OR prohledávání



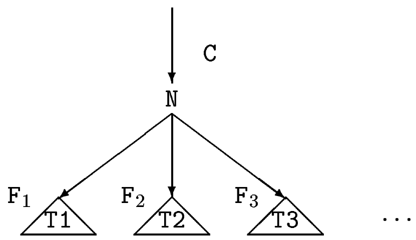
Obrázek 25: Příklad listu AND/OR stromu

3.8.3 Datová reprezentace AND/OR stromu

V následujícím si řekneme, jak budou vypadat data pro AND/OR prohledávání do hloubky. `N` budeme značit identifikátor uzlu, `C` bude cena hrany do uzlu `N`. `F` bude příslušná heuristická `F`-hodnota tohoto uzlu.

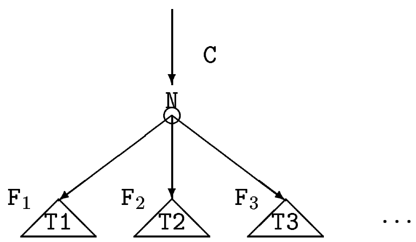
List AND/OR stromu (obr. 25) bude reprezentovat struktura `leaf(N,F,C)`. Příslušná hodnota `F = C + h(N)`.

OR-uzel AND/OR stromu (obr. 26) bude reprezentovat struktura `tree(N,F,C,or:[T1,T2,T3,...])`. Příslušná hodnota $\mathtt{F} = \mathtt{C} + \min_i \mathtt{F}_i$.



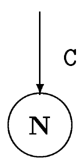
Obrázek 26: Příklad OR-uzlu AND/OR stromu

AND-uzel AND/OR stromu (obr. 27) bude reprezentovat struktura `tree(N,F,C,and:[T1,T2,T3,...])`. Příslušná hodnota $\mathtt{F} = \mathtt{C} + \sum_i \mathtt{F}_i$.



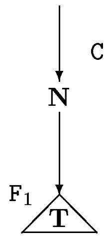
Obrázek 27: Příklad AND-uzlu AND/OR stromu

Rozřešený list AND/OR stromu (obr. 28) bude reprezentovat struktura `solvedleaf(N,F)` a příslušná hodnota heuristické funkce `F = C`.



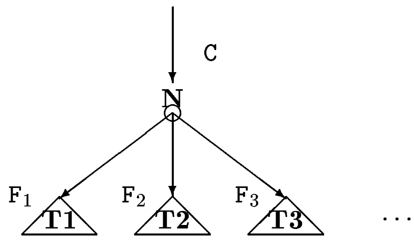
Obrázek 28: Příklad rozřešeného listu AND/OR stromu

Rozřešený OR-uzel AND/OR stromu (obr. 29) bude reprezentovat struktura `solvedtree(N,F,T)`, kde příslušná hodnota $\mathtt{F} = \mathtt{C} + \mathtt{F}_1$.



Obrázek 29: Příklad rozřešeného OR-uzlu AND/OR stromu

Rozřešený AND-uzel AND/OR stromu (obr. 30) bude reprezentován, jak již pilný čtenář tuší, strukturou tohoto tvaru: `solvedtree(N,F,and:[T1,T2,...])` a jeho příslušná hodnota $F = C + \sum_i F_i$.



Obrázek 30: Příklad rozřešeného AND-uzlu AND/OR stromu

3.8.4 Vlastní program

Program předpokládá existenci operátorů:

```
?- op(500,xfx,:).
?- op(600,xfx,-->).

andor(Node,Solution,Tree) :- expand(leaf(Node,0,0),9999,SolutionTree,yes).

%%%%%%%%% expand(Tree,Bound,NewTree,Solved). %%%%%%%%%%%%%%%%
expand(Tree,Bound,Tree,no) :- f(Tree,F),F>Bound,!.
expand(leaf(Node,F,C),_,solvedleaf(Node,F),yes) :- goal(Node),!.
expand(leaf(Node,F,C),Bound,NewTree,Solved) :- expandnode(Node,C,Tree1),!,
     expand(Tree1,Bound,NewTree,Solved);Solved=never,!.
expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved) :- Bound1 is Bound-C,
     expandlist(SubTrees,Bound1,NewSubs,Solved1),
     continue(Solved1,Node,C,NewSubs,Bound,NewTree,Solved).
expandlist(Trees,Bound,NewTrees,Solved) :-
     selecttree(Trees,Tree,OtherTrees,Bound,Bound1),
     expand(Tree,Bound1,NewTree,Solved1),
     combine(OtherTrees,NewTree,Solved1,NewTrees,Solved).
continue(yes,Node,C,SubTrees,_,solvedtree(Node,F,SubTrees),yes) :-
     backup(SubTrees,H), F is C+H,!.
continue(never,_,_,_,_,_,never) :- !.
continue(no,Node,C,SubTrees,Bound,NewTree,Solved) :- backup(SubTrees,H),
```

```
     F is C+H,!,expand(tree(Node,F,C,SubTrees),Bound,NewTree,Solved).
combine(or:_,Tree,yes,Tree,yes) :- !.
combine(or:Trees,Tree,no,or:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.
combine(or:[],_,never,_,never) :- !.
combine(or:Trees,_,never,Trees,no) :- !.
combine(and:Trees,Tree,yes,and:[Tree|Trees],yes) :- allsolved(Trees),!.
combine(and:_,_,never,_,never) :- !.
combine(and:Trees,Tree,YesNo,and:NewTrees,no) :- insert(Tree,Trees,NewTrees),!.
expandnode(Node,C,tree(Node,F,C,Op:SubTrees)) :-  Node --> Op:Successors,
     evaluate(Successors,SubTrees),backup(Op:SubTrees,H),F is C+H.
evaluate([],[]).
evaluate([Node/C|NodesCosts],Trees) :- h(Node,H),F is C+H,evaluate(NodesCosts,Tree1),
     insert(leaf(Node,F,C),Tree1,Trees).
allsolved([]).
allsolved([Tree|Trees]) :- solved(Tree),allsolved(Trees).
solved(solvedtree(_,_,_)).
solved(solvedleaf(_,_)).
f(Tree,F) :- arg(2,Tree,F),!.
insert(T,[],[T]) :- !.
insert(T,[T1|Ts],[T,T1|Ts]) :- solved(T1),!.
insert(T,[T1|Ts],[T1|Ts1]) :- solved(T),insert(T,Ts,Ts1),!.
insert(T,[T1|Ts],[T,T1|Ts]) :- f(T,F),f(T1,F1),F=<F1,!.
insert(T,[T1|Ts],[T1|Ts1]) :- insert(T,Ts,Ts1).
backup(or:[Tree|_],F) :- f(Tree,F),!.
backup(and:[],0) :- !.
backup(and:[Tree1|Trees],F) :- f(Tree1,F1),backup(and:Trees,F2),F is F1+F2,!.
backup(Tree,F) :- f(Tree,F).
selecttree(Op:[Tree],Tree,Op:[],Bound,Bound) :- !.
selecttree(Op:[Tree|Trees],Tree,Op:Trees,Bound,Bound1) :- backup(Op:Trees,F),
     (Op=or,!,min(Bound,F,Bound1);Op=and,Bound1 is Bound+F).
min(A,B,A) :- A<B,!.
min(A,B,B).
```

3.8.5 Hledání cesty mezi městy

Vraťme se ještě k příkladu uváděnému v odstavci 3.7.2. Reprezentace takové mapy by mohla vypadat tak, že každou spojnici dvou měst by reprezentoval predikát `s(City1,City2)`. Klíčové postavení města (např. postavení hraničního přechodu z odstavce 3.7.2) by reprezentoval predikát `key(City1-City2,City3)`, jehož sématika by měla tento význam:

> Město `City3` má klíčové postavení mezi městy `City1` a `City2`[9].

Vlastní vyhledávání by pak vypadalo takto:

1. Jsou-li `Y1, Y2,...` klíčové body mezi městy `A` a `Z`, pak hledej jednu z cest:
   - cestu z `A` do `Z` přes `Y1`
   - cestu z `A` do `Z` přes `Y2`
   - ...
2. Není-li mezi městy `A` a `Z` klíčové město, pak hledej souseda `Y` města `A` takového, že existuje cesta z `Y` do `Z`.

Teď si ukážeme, jak budeme konstruovat příslušný AND/OR strom. Budeme potřebovat operátory „`-`" a „`via`":

```
X-Z
X-Z via Y
?- op(560,xfx,via).
```

[9]Neboť každé z nich leží na jiné straně hranice.

AND/OR strom, který bude reprezentovat tento problém, bude vypadat takto:

```
X-Z --> or:Problemlist :- bagof((X-Z via Y)/0,key(X-Z,Y),Problemlist),!.
X-Z --> or:Problemlist :- bagof((Y-Z)/D,s(X,Y,D),Problemlist).
X-Z --> and:[(X-Y)/0,(Y-Z)/0].
goal(X-X).
/* h(Node,H).     ... heuristická funkce */
```

Platí, že když

$$\forall n \quad h(n) \leq h^*(n)$$

kde h^* je heuristika, která určí minimální cenu řešení uzlu n, pak najdeme vždy nejlepší řešení.

3.9 Algoritmy soupeřícího prohledávání

3.9.1 Minimax

Typickou úlohou prohledávání stavového prostoru se soupeřící strategií je hledání nejlepšího možného tahu v šachu, tzv. Minimax[10]. Každý tah je ohodnocený nějakou heuristickou funkcí, přičemž tento algoritmus vyhledává „optimální" tah. V závislosti na tom, zda „jsem na tahu já" resp. zda „je na tahu soupeř", vybírá algoritmus tah ohodnocený maximální resp. minimální hodnotou heuristické funkce.

Následující predikát `minmax` bude provádět prohledávání právě popsaným způsobem. Nebudeme zde definovat predikát `moves(Pos,PosList)`, který pro vstupní parametr `Pos` vrátí seznam možných pozící, do nichž se lze dostat z pozice `Pos` jedním tahem. Predikát `staticval(Pos,Val)` vrátí „statickou" hodnotu pozice, z níž již neexistuje možný tah, nebo z níž již nebudeme další tahy promýšlet[11]. Dále budeme definovat predikát `min_to_move(Pos)`, který uspěje, pokud „je na tahu soupeř", a `max_to_move(Pos)`, který uspěje, pokud „jsem na tahu já".

```
minmax(Pos,BestSucc,Val) :- moves(Pos,PosList),!,best(PosList,BestSucc,Val);
     staticval(Pos,Val).
best([Pos],Pos,Val) :- minmax(Pos,_,Val),!.
best([Pos1|PosList],BestPos,BestVal) :- minmax(Pos1,_,Val1),
     best(PosList,Pos2,Val2),betterof(Pos1,Val1,Pos2,Val2,BestPos,BestVal).
betterof(Pos0,Val0,Pos1,Val1,Pos0,Val0) :- min_to_move(Pos0),Val0>Val1,!;
     max_to_move(Pos0),Val0<Val1,!.
betterof(Pos0,Val0,Pos1,Val1,Pos1,Val1).
```

Poznamenejme na závěr, že `minmax(Pos,BestSucc,Val)` vezme pozici `Pos` a vrátí nejlepší možný tah `BestSucc`, jehož heuristická hodnota je `Val`.

3.9.2 Alfa-Beta procedura

Při provádění predikátu `minmax` se hodnota heuristické funkce pohybuje v nějakém intervalu $\langle \alpha, \beta \rangle$. Z toho plyne, že v průběhu provádění algoritmu můžeme „odříznout" podstromy, v nichž heuristická funkce „přeleze" β nebo „podleze" α.

Nyní tedy budeme počítat „zpětnou vazbu" `V(P)`, kde `P` je to, co vrátí predikát minmax. Tuto zpětnou vazbu budeme „zařezávat" podle následující tabulky:

V(P,Alpha,Beta) $\leq$ Alpha	if V(P) $\leq$ Alpha
V(P,Alpha,Beta) = V(P)	if Alpha < V(P) < Beta
V(P,Alpha,Beta) $\geq$ Beta	if V(P) $\geq$ Beta

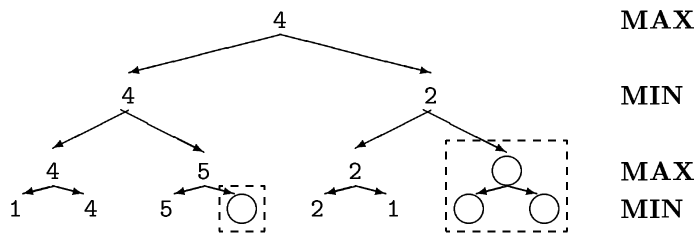
Když bychom počítali `V(P,`$-\infty$`,`∞`)`, dostaneme přesně `V(P)`.

[10] Zde se samozřejmě nebude jednat o známý hasící přístroj, ale o algoritmus, který pracuje na základě minimální resp. maximální hodnoty heuristické funkce. Správně by tedy měl být pojmenován „Minmax", ale „Minimax" má v sobě více šťávy.

[11] Predikát `staticval` slouží jinými slovy k tomu, aby byl splněn požadavek konečnosti prohledávacího stromu.

```
alphabeta(Pos,Alpha,Beta,GoodPos,Val) :- moves(Pos,PosList),!,
     boundedbest(PosList,Alpha,Beta,GoodPos,Val);staticval(Pos,Val).
boundedbest([Pos|PosList],Alpha,Beta,GoodPos,GoodVal) :-
     alphabeta(Pos,Alpha,Beta,_,Val),
     goodenough(PosList,Alpha,Beta,Pos,Val,GoodPos,GoodVal).
goodenough([],_,_,Pos,Val,Pos,Val) :- !.
goodenough(_,Alpha,Beta,Pos,Val,Pos,Val) :- min_to_move(Pos),Val > Beta,!;
     max_to_move(Pos),Val < Alpha,!.
goodenough(PosList,Alpha,Beta,Pos,Val,GoodPos,GoodVal) :-
     newbounds(Alpha,Beta,Pos,Val,NewAlpha,NewBeta),
     boundedbest(PosList,NewAlpha,NewBeta,Pos1,Val1),
     betterof(Pos,Val,Pos1,Val1,GoodPos,GoodVal).
newbounds(Alpha,Beta,Pos,Val,Val,Beta) :- min_to_move(Pos),Val > Alpha,!.
newbounds(Alpha,Beta,Pos,Val,Alpha,Val) :- max_to_move(Pos),Val < Beta,!.
newbounds(Alpha,Beta,_,_,Alpha,Beta).
betterof(Pos,Val,Pos1,Val1,Pos,Val) :- min_to_move(Pos),Val > Val1,!;
     max_to_move(Pos),Val < Val1,!.
betterof(Pos,Val,Pos1,Val1,Pos1,Val1).
```

Obrázek 31 ukazuje příklad stromu, který zpracuje predikát `minmax`. V čárkovaných rámečcích jsou podstromy, které „zařízne" predikát `alphabeta`, neboť na jejich heuristických hodnotách vůbec nezáleží. Proto je alfa-beta procedura efektivnější variantou „minimaxu".

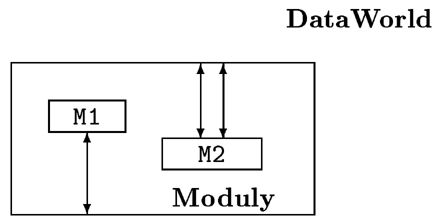


Obrázek 31: Zaříznutí Alfa-Beta procedurou

4 Expertní systémy

4.1 Pattern–directed programming

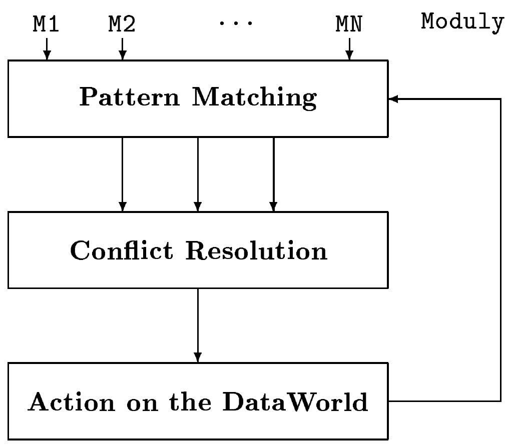
Motivující obrázek k tzv. *pattern–directed* programování ukazuje obr. 32. Množinu tzv. *modulů* obklopuje tzv. *svět dat* (*dataworld*). Tyto moduly mají schopnost – v případě jejich aktivace – zasáhnout a změnit dataworld (provést operaci nad daty). Způsob uplatnění jednotlivých modulů ukazuje obr. 33.



Obrázek 32: Motivace k pattern–directed programming

Pattern Matching vybere z množiny modulů `{M1,M2,...,MN}` ty moduly, které vyhovují určitému „vzoru”, t.j. které lze uplatnit vzhledem k aktuálnímu stavu dat. Pattern Matching nemusí vybrat pouze jeden uplatnění schopný modul; může vybrat několik, ba dokonce všechny moduly.

Conflict Resolution je prostředek, který ze vhodných modulů vybere právě jeden (nejlépe ten nejvýhodnější), který je nakonec uplatněn a provede svoji `Action` nad daty.



Obrázek 33: Diagram uplatňování modulů

Jednotlivé moduly budeme zapisovat:

```
ConditionPart --> ActionPart
[Condition1,Condition2,...] --> [Action1,Action2,...]
```

K tomu si musíme definovat operátor `-->`:

```
?- op(800,xfx,-->).
```

Algoritmus, který bude realizovat přístup pattern-directed programming, je následující:

```
run :- Condition --> Action,test(Condition),execute(Action).
test([]).
test([First|Rest]) :- call(First),test(Rest).
execute([stop]) :- !.
execute([]) :- run.
execute([First|Rest]) :- call(First),execute(Rest).
replace(A,B) :- retract(A),!,assert(B).
```

Uveďme si nyní příklad, jak lze jednoduchý matematický problém – nalezení největšího společného dělitele dvou čísel – řešit pomocí pattern–directed programming.

Klasická specifikace nalezení největšího společného dělitele dvou čísel zní:

```
pokud A ≠ B opakuj
     jestliže A > B pak nahraď číslo A číslem A − B
     jinak nahraď B číslem B − A
největší_společný_dělitel je A    /* nebo B */
```

Pravidla, která pomocí v tomto odstavci již uvedeného predikátu `run` budou hledat NSD dvou čísel, pak mohou vypadat takto:

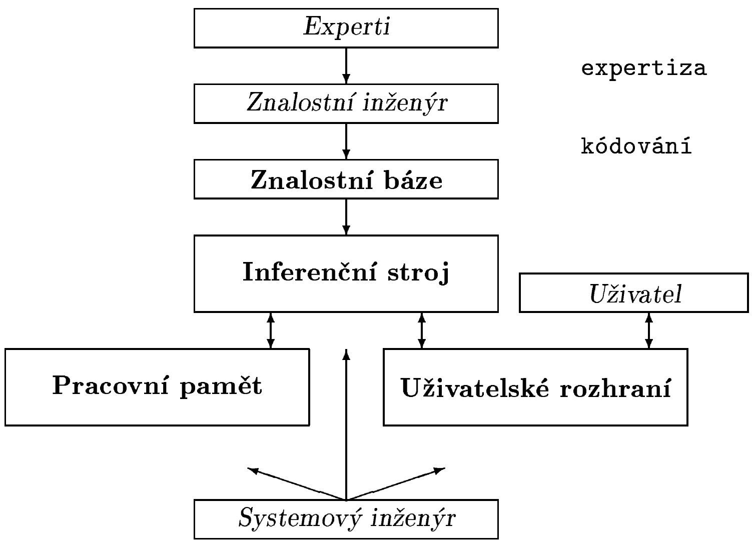
```
[number X,number Y,X > Y] --> [NewX is X-Y,replace(number X,number NewX)].
[number X] --> [write(X),stop].
```

Poznamenejme ještě, že operátor `number` je definován jako:

```
?- op(300,fx,number).
```

4.2 Struktura expertního systému

Strukturu expertního systému, jeho vzniku a provozování ukazuje obr. 34. *Experti* z dané oblasti dají hlavy dohromady



Obrázek 34: Struktura expertního systému

a vypadne z nich soubor znalostí – tzv. *expertiza*. Na tu sedne *znalostní inženýr*, který ji schroustá a vypadne z něj *znalostní báze*. Tu pak bude používat *inferenční stroj* k tomu, aby mohl radit uživatelům. Provozní součásti expertního systému, tedy *inferenční stroj*, *pracovní paměť* a *uživatelské rozhraní* má na starosti *systémový inženýr*.

4.3 Dopředné a zpětné řetězení

4.3.1 Druhy pravidel

Pravidla v pattern–directed programech mohou být dvojího druhu:

- logické pravidlo (diagnóza), např.:

```
IF family is albatros and color is dark
   THEN bird is black footed albatros.
```

- operativní pravidlo, např.:

```
IF unplaced tv and couch on WALL(X) and WALL(Y) is opposite WALL(X)
   THEN place tv on WALL(Y).
```

4.3.2 Dopředné a zpětné řetězení

V teorii expertních systémů rozeznáváme dvě strategie zpracování dat.

- dopředné řetězení:

```
Data ⟶ Rules ⟶ Conclusion
```

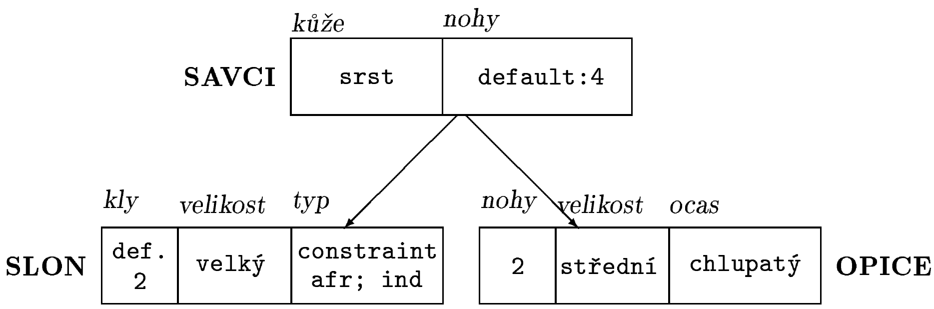
- zpětné řetězení:

```
Subgoals ⟵ Rules ⟵ Goal
```

Tyto strategie jsou však ve většině reálných expertních systémů kombinovány podle druhu dat a také kvůli efektivnosti.

4.3.3 Ukládání dat v expertních systémech

Data v expertních systémech ukládáme v různých více či méně „inteligentních" strukturách, jako jsou fakty, relace, záznamy,... Jednou z inteligentnějších struktur jsou tzv. **rámce**. Obsahují jakési rysy, které známe z objektově orientovaných programovacích jazyků, zejména pak *dědičnost*. Příklad uchování dat o savcích, slonovi a opici ukazuje obr. 35.



Obrázek 35: Příklad struktury „rámců"

5 Zpětné řetězení

5.1 Naivní expertní systém

Zpětné řetězení v podstatě spočívá v tom, že uživatel zadá nějakou hypotézu a expertní systém se snaží rozkládat tuto hypotézu na podproblémy tak dlouho, dokud to nejsou elementární fakta, která zná, nebo elementární fakta, na která se lze zeptat uživatele. Tímto způsobem tedy vzniká strom, složený z:

- kořenových uzlů, které reprezentují hypotézy (Goals),
- mezilehlých uzlů, které reprezentují dílčí hypotézy,
- a listových uzlů, které reprezentují elementární fakta a otázky uživateli.

Příklad takových uzlů:

```
k(1) :- m(1),m(3),l(2).
k(1,první_hypotéza).
m(1) :- beg(1),(l(1);l(5)),end(1),!.
m(3) :- beg(3),l(4),l(7),m(2),end(3),!.
m(3,dílčí_hypotéza_3).
l(2,otázka_2).
```

Dialog s uživatelem by pak mohl vypadat takto:

```
l(X) :- l(X,Ot), repeat,write(Ot),read(Odp),
     (Odp=a,asserta(l(X) :- !);Odp=n,asserta(l(X) :- !,fail)),!,
     fail;fail),!.
beg(X) :- asserta((m(X) :- !,fail)).
end(X) :- asserta(m(X) :- !).
```

Tento „naivní" expertní systém nepokládá dvakrát tutéž otázku a nepočítá dvakrát tytéž mezilehlé uzly.

5.1.1 Ukládání dat

Data se dají ukládat takto:

```
atribut(hodnota)
```

např.:

```
region(morava)
```

5.1.2 Dotazy uživateli

Chceme-li se zeptat uživatele na barvu

```
color(X) :- ask(color,X).
```

K uložení informací budeme používat predikát `know(YesNo,Attr,Val)`. Takže predikát `ask` můžeme psát:

```
ask(A,V) :- know(yes,A,V),!.
ask(A,V) :- know(_,A,V),!,fail.
ask(A,V) :- write(A:V),write('?:'),read(Y),asserta(know(Y,A,V)),Y==yes.
```

5.1.3 Vícehodnotové odpovědi

Je-li atribut A vícehodnotový, pak tato informace bude zaznamenána v programové databázi faktem:

```
multivaued(A).
```

K proceduře ask potom ještě přiřadíme klauzuli:

```
ask(A,V) :- not multivalued(A),know(yes,A,V2),V \== V2,!,fail.
```

Nyní si ještě pro příklad uveďme dotaz na velikost:

```
size(X) :- menuask(size,X,[large,plump,medium,small]).
menuask(A,V,Menuask) :- write('What is the value for '),write(A),
     write('?'),nl,write(Menuask),nl,read(X),check_val(X,A,V,Menuask),
     asserta(know(yes,A,X)),X == V.
check_val(X,A,V,MenuList) :- member(X,MenuList),!.
check_val(X,A,V,MenuList) :- write(X),write(' is not legal value. Try again!'),
     nl,menuask(A,V,MenuList).
```

5.2 Jednoduchý „shell"

```
top_goal(X) :- bird(X).
solve :- abolish(know,3),top_goal(X),write('The answer is '),write(X),nl.
solve :- write('No answer found.'),nl.
go :- greeting,repeat,write('>'),read(X),do(X),X == quit.
greeting :- write('Enter load, consult or quit at the prompt.'),nl.
do(load) :- load_kb,!.
do(consult) :- solve,!.
do(quit).
do(X) :- write(X),write(' is not legal command.'),nl,fail.
bird(black_footed_albatros) :- family(albatros),color(dark).
family(albatros) :- order(tubemosa),size(large),wings(long_narrow).
size(X) :- ask(size,X).
size(X) :- menuask(size,X,[large,plump,medium,small]).
```

5.3 Faktor jistoty

Někdy uživatel neodhalí přesně veškeré niance mezi povolenými hodnotami atributu (u velikosti třeba nepozná, co je přesně myšleno pojmem `velký` a co pojmem `středně velký` a jaký je mezi tím rozdíl). Proto zavádíme tzv. *faktor jistoty* (značíme `cf` a počítáme s ním v procentech).

Samozřejmě, nemusíme ohodnocovat faktorem jistoty jen odpovědi uživatele, ale i pravidla. To proto, že toto pravidlo sice platí, ale ne úplně vždy. Pokud bychom chtěli použít k dedukci pravidlo

```
IF byl jsem dnes v práci THEN je Po, Út, St, Čt nebo Pá
```

což je ve valné většině většině případů zcela regulerní. Ještě nedávno se však často stávalo, že jsme šli na 1. máje do průvodu a tuto spontánní radost potom nadpracovávali následující sobotu. To znamená, že nelze použít toto pravidlo k dedukci jen tak, ale s přihlédnutím k tomu, že nemusí být korektní. Proto můžeme každému pravidlu přiřknout také jistý faktor jistoty.

Tím nám v psaní expertního systému přibudou problémy navíc. Zejména:

- Jakou hodnotu `cf` stanovit jako prahovou[12] pro použití pravidla v dedukci.
- Jak spočítat ze známého `cf` předpokladů a známého `cf` pravidla správný `cf` závěru.

Také musíme vyřešit problém, jak vypočítat `cf` pro B v případech:

[12] prahovou hodnotu, t.j. minimum pro to, aby bylo pravidlo použito pro předpoklad; většinou se používá hodnoty 20%

- Pravidlo `if A then B` má $\texttt{cf}_\texttt{p}$ roven 50 a předpoklad `A` má $\texttt{cf}_\texttt{A}$ roven 100[13].
- Pravidlo `if A then B` má `cf` roven 50 a předpoklad `A` má `cf` roven 80.
- Pravidlo `if A` $\wedge$ `B then C` má $\texttt{cf}_\texttt{p}$ a předpoklady `A` resp. `B` mají $\texttt{cf}_\texttt{A}$ resp. $\texttt{cf}_\texttt{B}$. Jak vypočítat $\texttt{cf}_\texttt{C}$ závěru `C`?

Tyto problémy si většinou vyřeší autor expertního systému podle svého, protože neexistuje žádné na 100% ověřené řešení, o kterém by bylo dokázáno, že je nejsprávnější. Proto si zde uvedeme jedno z mnoha možných a používaných.

- $\texttt{cf}_\texttt{A,B}$ konjunkce předpokladů `A`, `B` s faktory jistoty $\texttt{cf}_\texttt{A}$, $\texttt{cf}_\texttt{B}$ spočteme:

$$\texttt{cf}_\texttt{A,B} = \min(\texttt{cf}_\texttt{A}, \texttt{cf}_\texttt{B})$$

- $\texttt{cf}_\texttt{Z}$ závěru z faktoru jistoty pravidla $\texttt{cf}_\texttt{P}$ a faktoru jistoty předpokladů $\texttt{cf}_\texttt{A,B}$ spočteme:

$$\texttt{cf}_\texttt{Z} = \frac{\texttt{cf}_\texttt{P} * \texttt{cf}_\texttt{A,B}}{100}$$

5.3.1 Kombinování faktoru jistoty

V našem expertním systému s faktorem jistoty však budeme používat faktor jistoty s oborem hodnot v intervalu $< -100, 100 >$. Tento postup použili též autoři úspěšného expertního systému v oblasti mediciny „**Mycin**". Hodnota 100 zde reprezentuje stoprocentní jistotu pozitivní („určite ano"), hodnota -100 pak stoprocentní jistotu negativní („určitě ne").

Pro kombinaci dvou hodnot `cf` pak musíme použít tyto vzorce:

- $\texttt{cf}(X,Y) = X + \frac{Y(100-X)}{100}$, pokud $X, Y \geq 0$.
- $\texttt{cf}(X,Y) = 100\frac{X+Y}{100} - \min(|X|,|Y|)$, pokud $(X < 0) \vee (Y < 0)$.
- $\texttt{cf}(X,Y) = -\left(-X - Y\frac{100+X}{100}\right)$, pokud $X, Y < 0$.

5.3.2 Uchovávání dat

Pro implementaci expertního systému se zpětným řetězením a faktorem jistoty budeme uchovávat tato data:

- `rule(Name,LHS,RHS).` ... pravidlo, kde `LHS` je levá strana a `RHS` pravá strana pravidla, tedy tento predikát reprezentuje pravidlo:

```
Name: IF LHS THEN RHS
```

- `rhs(Goal,CF).` ... pravá strana pravidla.
- `lhs(GoalList).` ... levá strana pravidla.
- `av(Attr,Val).` ... reprezentuje hodnotu `Val` atributu `Attr`.
- `fact(av(Attr,Val),CF).` ... takto budeme ukládat do programové databáze známé podcíle.

5.3.3 Zdrojový text

Zde si nyní nadefinujeme predikát `findgoal(av(problem,X),CF)`, který vyřeší `problem` a vrátí jeho hodnotu `X` a faktor jistoty `CF`. Tento predikát uspěje, pokud je hodnota atributu

- buď známa,
- nebo odvozena z pravidel,
- nebo je třeba položit otázku uživateli – v tom případě uspěje predikát `askable(atribut,'Otázka')`.

[13]Toto se obvykle řeší jednoduše tak, že pokud předpoklady platí s `cf` = 100, pak závěr pravidla získá tentýž `cf`, jako má pravidlo.

V rámci tohoto predikátu budeme též definovat predikáty:

- `prove`, který prověří `lhs` a najde `CF`,
- `adjust`, který zkombinuje `lhs`, `rhs` a `CF`,
- `update`, který obohatí paměť o nové závěry[14].

```
findgoal(av(Attr,Val),CF) :- fact(av(Attr,Val),CF),!.
findgoal(av(Attr,Val),CF) :- not fact(av(Attr,_),_),askable(Attr,Prompt),
     query_user(Attr,Prompt),!,findgoal(av(Attr,Val),CF).
query_user(Attr,Prompt) :- write(Prompt),read(Val),read(CF),
     asserta(fact(av(Attr,Val),CF)).
findgoal(Goal,CurCF) :- fg(Goal,CurCF).    /* zkrácení názvu */
fg(Goal,CurCF) :- rule(N,lhs(IfList),rhs(Goal,CF)),prove(IfList,Tally),
     adjust(CF,Tally,NewCF),update(Goal,NewCF,CurCF),CurCF == 100,!.
fg(Goal,CF) :- fact(Goal,CF).
prove(IfList,Tally) :- prov(IfList,100,Tally).
prov([],Tally,Tally).
prov([H|T],CurTal,Tally) :- findgoal(H,CF),min(CurTal,CF,Tal),Tal >= 20,
     prov(T,Tal,Tally).
min(X,Y,X) :- X =< Y,!.
min(X,Y,Y) :- Y =< X.
adjust(CF1,CF2,CF) :- X is CF1 * CF2 / 100,int_round(X,CF).
int_round(X,I) :- X >= 0,I is integer(X + 0.5).
int_round(X,I) :- X < 0,I is integer(X - 0.5).
update(Goal,NewCF,CF) :- fact(Goal,OldCF),combine(NewCF,OldCF,CF),
     retract(fact(Goal,OldCF)),asserta(fact(Goal,CF)),!.
update(Goal,CF,CF) :- asserta(fact(Goal,CF)).
combine(CF1,CF2,CF) :- CF1 >= 0,CF2 >= 0,X is CF1 + CF2 * (100 - CF1)/100,
     int_round(X,CF).
combine(CF1,CF2,CF) :- CF1 < 0,CF2 < 0,X is -(-CF1 - CF2 * (100 + CF1)/100),
     int_round(X,CF).
combine(CF1,CF2,CF) :- (CF1 < 0;CF2 < 0),(CF1 > 0;CF2 > 0),abs_min(CF1,CF2,MCF),
     X is 100 * (CF1 + CF2)/(100 - MCF),int_round(X,CF).
```

Pokud bychom chtěli znát odpověď na negovaný cíl, pak to provedeme následovně:

```
findgoal(not Goal,NCF) :- findgoal(Goal,CF),NCF is -CF,!.
```

Tady asi ještě něco chybí!!

5.3.4 Super shell

Pro výše specifikovaný predikát `findgoal/2` můžeme nyní definovat uživatelské rozhraní, které pracovně nazveme *super shell*.

```
super :- repeat,write('consult,load,exit'),nl,write(':'),read_line(X),
     doit(X),X == exit.
doit(consult) :- top_goals,!.
doit(load) :- load_rules,!.
doit(exit).
top_goals :- top_goal(Attr),top(Attr),print_goal(Attr),fail.
top_goals.
top(Attr) :- findgoal(av(Attr,Val),CF),!.
top(_) :- true.
```

[14] Závěry XVII. sjezdu splníme a nepřekazí nám to žádné imperialistické rejdy :–)

```
print_goal(Attr) :- nl,fact(av(Attr,X),CF),CF >= 20,outp(av(Attr,X),CF),nl,fail.
print_goal(Attr) :- write('done with '),write(Attr),nl.
outp(av(A,V),CF) :- output(A,V,PrintList),write(A - 'cf' - CF), printlist(PrintList),!.
outp(av(A,V),CF) :- write(A - V - 'cf' - CF).
printlist([]).
printlist([H|T]) :- write(H),printlist(T).
```

5.4 Trasování expertního systému

Někdy je potřeba (zejména pro „ladění” pravidel znalostním inženýrem) uchovávat informace o běhu expertního systému. Jak náš expertní systém z odstavce 5.3.3 modifikovat, aby splňoval tento požadavek si ukážeme v tomto odstavci.

K tomuto účelu si nejprve nadefinujeme potřebné predikáty `bugdisp`, který vypíše „trasovací hlášku” a `set_trace`, který vypne nebo zapne trasovací režim expertního systému. Dále ještě obohatíme super shell o příkaz `trace(on/off)` tím, že přidáme jedno pravidlo predikátu `doit`.

```
bugdisp(L) :- ruletrace,write_line(L),!.
bugdisp(_).
write_line([]) :- nl.
write_line([H|T]) :- write(H),tab(1),write_line(T).
doit(trace(X)) :- set_trace(X),!.
set_trace(off) :- ruletrace,retract(ruletrace).
set_trace(on) :- not ruletrace,asserta(ruletrace).
set_trace(_).
```

Nyní si uvedeme, jaké změny musíme provést ve zdrojovém textu z odstavce 5.3.3. Následující zdrojový text je v podstatě ten samý jako v odstavci 5.3.3 a úseky do něj přidané jsou *proloženy*.

```
findgoal(av(Attr,Val),CF) :- fact(av(Attr,Val),CF),!.
findgoal(av(Attr,Val),CF) :- not fact(av(Attr,_),_),askable(Attr,Prompt),
     query_user(Attr,Prompt),!,findgoal(av(Attr,Val),CF).
query_user(Attr,Prompt) :- write(Prompt),read(Val),read(CF),
     asserta(fact(av(Attr,Val),CF)).
findgoal(Goal,CurCF) :- fg(Goal,CurCF).    /* zkrácení názvu */
fg(Goal,CurCF) :- rule(N,lhs(IfList),rhs(Goal,CF)), bugdisp(['Call rule',N]),
     prove(IfList,Tally), bugdisp(['Exit rule',N]),
     adjust(CF,Tally,NewCF),update(Goal,NewCF,CurCF),CurCF == 100,!.
fg(Goal,CF) :- fact(Goal,CF).
prove(IfList,Tally) :- prov(IfList,100,Tally).
prove(N,_) :- bugdisp(['Fail rule',N]),fail.
prov([],Tally,Tally).
prov([H|T],CurTal,Tally) :- findgoal(H,CF),min(CurTal,CF,Tal),Tal >= 20,
     prov(T,Tal,Tally).
min(X,Y,X) :- X =< Y,!.
min(X,Y,Y) :- Y =< X.
adjust(CF1,CF2,CF) :- X is CF1 * CF2 / 100,int_round(X,CF).
int_round(X,I) :- X >= 0,I is integer(X + 0.5).
int_round(X,I) :- X < 0,I is integer(X - 0.5).
update(Goal,NewCF,CF) :- fact(Goal,OldCF),combine(NewCF,OldCF,CF),
     retract(fact(Goal,OldCF)),asserta(fact(Goal,CF)),!.
update(Goal,CF,CF) :- asserta(fact(Goal,CF)).
combine(CF1,CF2,CF) :- CF1 >= 0,CF2 >= 0,X is CF1 + CF2 * (100 - CF1)/100,
     int_round(X,CF).
combine(CF1,CF2,CF) :- CF1 < 0,CF2 < 0,X is -(-CF1 - CF2 * (100 + CF1)/100),
     int_round(X,CF).
```

```
combine(CF1,CF2,CF) :- (CF1 < 0;CF2 < 0),(CF1 > 0;CF2 > 0),abs_min(CF1,CF2,MCF),
     X is 100 * (CF1 + CF2)/(100 - MCF),int_round(X,CF).
```

5.5 Způsob získání závěru

Někdy je také nutné, aby expertní systém uměl odpovědět nejen na danou otázku, ale také aby vysvětlil „svůj myšlenkový postup". V řeči zpětného řetězení to znamená s příslušnou odpovědí vrátit i cestu ve stromě zpětného řetězení, vedoucí k řešení problému.

Za tímto účelem musíme „rozšířit" predikát `fact/2` na `fact/3`, kde třetím parametrem bude právě příslušná cesta k tomuto faktu:

```
fact(AV,CF,RuleList).
```

K tomuto cíli stačí opět jen zmodifikovat již vytvožený predikát `update`, který bude do programové databáze zařazovat náš předefinovaný predikát `fact/3`. Přidané fragmenty zdrojového textu jsou opět *proloženy*.

```
update(Goal,NewCF,CF, RuleN) :- fact(Goal,OldCF, OldRules),
     combine(NewCF,OldCF,CF),retract(fact(Goal,OldCF, OldRules)),
     asserta(fact(Goal,CF, [RuleN| OldRules])),!.
update(Goal,CF,CF, RuleN) :- asserta(fact(Goal,CF, [RuleN])).
```

Po proběhnutí takto modifikovaného expertního systému z odstavce 5.3.3 pak můžeme obdržet odpověď a způsob jejího dosažení po zavolání tohoto predikátu `how`.

```
how(Goal) :- fact(Goal,CF,Rules),CF > 20,pretty(Goal,PG),
     write_line([PG,was,derrived,from,'rules:'|Rules]),nl,list_rules(Rules),fail.
how(_).
how(not Goal) :- fact(Goal,CF,Rules),CF < -20,pretty(not Goal,PG),
     write_line([PG,was,derrived,from,'rules:'|Rules]),nl,list_rules(Rules),fail.
pretty(av(A,yes),[A]) :- !.
pretty(not av(A,yes),[not,A]) :- !.
pretty(av(A,no),[not,A]) :- !.
pretty(not av(A,V),[not,A,is,V]).
pretty(av(A,V),[A,is,V]).
list_rules([]).
list_rules([R|X]) :- list_rule(R),list_rules(X).
list_rule(N) :- rule(N,lhs(IfList),rhs(Goal,CF)),write_line(['rule ',N]),
     write_line(['If']),write_ifs(IfList),write_line(['Then']),
     pretty(Goal,PG),write_line([' ',PG,CF]),nl.
write_ifs([]).
write_ifs([H|T]) :- pretty(H,HP),tab(5),write_line(HP),write_ifs(T).
```

V případě, že bychom chtěli, aby predikát `how` uměl komunikovat s uživatelem, stačí jej napsat takto:

```
how :- write('Goal? '),read_line(X),nl,pretty(Goal,X),how(Goal).
```

Můžeme však predikát `how` začlenit takřka okamžitě do super shellu, když do něj přidáme klauzuli:

```
doit(how) :- how,!.
```

5.6 Zdůvodnění získání závěru

Někdy také můžeme na expertní systém klást ještě další požadavek. Nemusí nám stačit cesta k řešení, ale také historie, kterou expertní systém v době svého výpočtu prošel – tedy nejen úspěšně aplikovaná pravidla, ale i ta, která skončila neúspěchem (`fail`).

```
findgoal(Goal,CurCF, Hist) :- fg(Goal,CurCF, Hist).
fg(Goal,CurCF, Hist) :- rule(N,lhs(IfList),rhs(Goal,CF)),
     prove( N,IfList,Tally, Hist),adjust(CF,Tally,NewCF),
     update(Goal,NewCF,CurCF),CurCF == 100,!.
prove( N,IfList,Tally, Hist) :- prov(IfList,100,Tally, [N|Hist]), !.
prove(N,_,_,_) :- bugdisp(['Fail rule',N]),fail.
prov([],Tally,Tally, Hist).
prov([H|T],CurTal,Tally, Hist) :- findgoal(H,CF, Hist),min(CurTal,CF,Tal),Tal >= 20,
     prov(T,Tal,Tally, Hist).
```

Ještě nám zbývá dořešit problém, jak vypsat historii.

```
get_user(X,Hist) :- repeat,write(':'),read_line(X),process_ans(X,Hist).
process_ans([why],Hist) :- nl,write_hist(Hist),!,fail.
process_ans([trace,X],_) :- set_trace(X),!,fail.
process_ans([help],_) :- help,!,fail.
process_ans(X,_).
write_hist([]) :- nl.
write_hist([goal(X)|T]) :- write_line([goal,X]),!,write_hist(T).
write_hist([N|T]) :- list_rule(N),!,write_hist(T).
```

6 Dopředné řetězení

6.1 Principy

V *pracovní paměti* expertního systému s dopředným řetězením máme pravidla tvaru:

left hand side (LHS) $\Longrightarrow$ right hand side (RHS)

kde LHS je soubor podmínek a RHS je soubor akcí. Na tato pravidla pak opakovaně aplikujeme následující postup:

1. Výběr pravidla, jehož LHS uspěje se současným stavem pracovní paměti;
2. Provedení RHS pravidla (což většinou mění stav pracovní paměti);
3. Opakovat body 1 a 2, dokud to lze.

V Prologu budeme pracovní paměť dopředného řetězení implementovat v programové databázi, kam budeme ukládat pravidla v tomto tvaru:

```
rule <rule_id>:[<N>:<condition>,...] ==> [<action>,...].
```

Příklad 6.1

```
rule id6: [1: has(X,pointed_teeth),2: has(X,claws),3: has(X,forward_eyes)]
     ==> [retract(all),assert(isa(X,carnivore))].
rule id10: [1: isa(X,mammal),2: isa(X,carnivore),3:has(X,black_stripes)]
     ==> [retract(all),assert(isa(X,tiger))].
rule id16: [1: isa(Animal,Type),2: parent(Animal,Child)]
     ==> [retract(2),assert(isa(Child,Type))].
```

Inicializační soubor dat do pracovní paměti expertního systému s dopředným řetězením můžeme zapsat zavoláním predikátu `initial_data([<term>,...]`, např.:

```
initial_data([gives(robie,milk),eats_meat(robie),...]).
initial_data([read_facts]).
```

6.2 Ilustrativní příklad s rozestavováním nábytku

V tomto odstavci si uvedeme příklad pravidel pro rozestavování nábytku v místnosti.

```
rule 1:[1: end,2: read_facts] ==> [retract(all)].
rule 2:[1: read_facts] ==> [prompt('Attribute? ',X),assert(X)].
rule f1:[1: furniture(couch,LenC),position(door,DoorWall),
          opposite(DoorWall,OW),right(DoorWall,RW),
          2: wall(OW,LenOW),wall(RW,LenRW),LenOW >= LenRW,LenC =< LenOW]
     ==> [retract(1),assert(position(couch,OW)),retract(2),
          NewSpace = LenOW - LenC,assert(wall(OW,NewSpace))].
rule f3:[1: furniture(tv,LenTv),2: position(couch,CW),3: opposite(CW,W),
          4: wall(W,LenW),LenW >= LenTv]
     ==> [retract(1),assert(position(tv,W)),retract(4),NewSpace = LenW - LenTv,
          assert(wall(W,NewSpace))].
```

6.3 Expertní systém s dopředným řetězením

V tomto odstavci uvedeme implementaci expertního systému s dopředným řetězením. K tomu budeme potřebovat následující operátory:

```
?- op(230,xfx,==>).
?- op(32,xfy,:).
?- op(250,fx,rule).
```

K těmto operátorům ještě připojíme inicializační data, která budou sloužit k demonstraci tohoto expertního systému.

```
?- asserta(fact(isa(robie,carnivore))).
```

A nyní již slibovaný zdrojový text:

```
/*  dopředné řetězení */
go :- call(rule(ID:LHS ==> RHS),try(LHS,RHS),write('Rule fired '),write(ID),nl,!,go.
go :- nl,write(done),nl,print_state.
try(LHS,RHS) :- match(LHS),process(RHS,LHS).
match([]) :- !.
match([N:Prem|Rest]) :- !,(fact(Prem);test(Prem)),match(Rest).
match([Prem|Rest]) :- (fact(Prem);test(Prem)),match(Rest).
test(X >= Y) :- X >= Y,!.
test(X = Y) :- X is Y,!.
test(X # Y) :- X = Y,!.
test(member(X,Y)) :- member(X,Y),!.
test(not (X)) :- fact(X),!,fail.
process([],_) :- !.
process([Action|Rest],LHS) :- take(Action,LHS),process(Rest,LHS).
take(retract(N),LHS) :- (N == all;integer(N)),retr(N,LHS),!.
take(A,_) :- take(A).
take(retract(X)) :- retract(fact(X)),!.
take(assert(X)) :- asserta(fact(X)),write(adding_X),nl,!.
take(X # Y) :- X = Y,!.
take(X = Y) :- X is Y,!.
take(write(X)) :- write(X),!.
take(nl) :- nl,!.
take(read(X)) :- read(X),!.
retr(all,LHS) :- retrall(LHS),!.
retr(N,[]) :- write('retract error, no'|N),nl,!.
retr(N,[N:Prem|_]) :- retract(fact(Prem)),!.
retr(N,[_|Rest]) :- !,retr(N,Rest).
retrall([]).
retrall([N:Prem|Rest]) :- retract(fact(Prem)),!,retrall(Rest).
retrall([_|Rest]) :- retrall(Rest).
```

6.4 Generování konfliktních pravidel

6.4.1 Konfliktní pravidla a jejich vznik

V době běhu expertního systému s dopředným řetězením může stát, že při pattern matching může levé straně určitého pravidla odpovídat více faktů v pracovní paměti. Např. „matchuje-li" expertní systém pravidlo

```
rule 12:[...,eats(X,meat),...] ==> ...
```

a při tom v pracovní paměti jsou tato fakta:

```
eats(robie,meat).
eats(suzie,meat).
```

pak musíme ještě v průběhu tohoto matchingu určit, zdali se má proměnná `X` unifikovat s atomem `robie` či s atomem `suzie`. Proto je nutné najít způsob, jak tuto *konfliktní množinu* vyřešit.

V Prologu je (někdy) vestavěný predikát `findall`, který má tuto definici:

```
findall(X,Goal,XList) :- call(Goal),assertz(stack(X)),fail;assertz(stack(bottom)),
     collect(XList).
collect(L) :- retract(stack(X)),!,(X = bottom,!,L=[];L=[X|Rest],collect(Rest)).
```

Abychom mohli lépe manipulovat s pravidly, zavedeme si predikát `r` s aritou 4. Predikát `conflict_set` zkonstruuje konfliktní množinu:

```
conflict_set(CS) :- findall(r(Inst,ID,LHS,RHS),(rule ID:LHS ==> RHS,match(LHS,Inst)),CS).
```

Jak jsem uvedli výše, je nutné konfliktní množinu vyřešit během matchingu. Proto musíme modifikovat predikát `match`.

```
match([],[]) :- !.
match([N:Prem|Rest],[Prem/Time|IRest]) :- !,(fact(Prem,Time);test(Prem),Time=0),
     match(Rest,IRest).
match([Prem|Rest],[Prem/Time|IRest]) :- (fact(Prem,Time);test(Prem),Time=0),
     match(Rest,IRest).
assert_ws(fact(X,T)) :- getchron(T),asserta(fact(X,T)).
getchron(N) :- retract(chron(N),NN is N+1,asserta(chron(NN)),!.
```

Pomocí takto modifikovaného predikátu `match` navíc ke každé položce `LHS` přidáme čas jejího zařazení (tzn. počet „cyklů" expertního systému, které proběhly od zařazení faktu do pracovní paměti).

6.4.2 LEX metoda

Princip této metody spočívá v:

1. odkládání určité použité instance,
2. upřednostňování pravidla užívajícího nejčerstvější fakta,
3. upřednostňování „specifikovanějších" faktů.

```
go :- conflict_set(CS),select_rule(CS,r(Inst,ID,LHS,RHS)),
     process(RHS,LHS),asserta(instantiation(Inst)),write('Rule fired '),
     write(ID),nl,!,go.
select_rule(CS,R) :- refract(CS,CS1),lex_sort(CS1,[R|_]).
refract([],[]).
refract([r(Inst,_,_,_)|T],TR) :- instantiation(Inst),!,refract(T,TR).
refract([H|T],[H|TR]) :- refract(T,TR).
lex_sort(L,R) :- build_keys(L,LK),keysort(LK,X),reverse(X,Y),strip_keys(Y,R).
build_keys([],[]).
build_keys([r(Inst,A,B,C)|T],[Key-r(Inst,A,B,C)|TR]) :- build_chlist(Inst,ChL),
     sort(ChL,X),reverse(X,Key),build_keys(T,TR).
build_chlist([],[]).
build_chlist([_/Chron|T],[Chron|TC]) :- build_chlist(T,TC).
strip_keys([],[]).
strip_keys([Key-X|Y],[X|Z]) :- strip_keys(Y,Z).
```

Ve výše uvedeném zdrojovém textu nám chybí ještě dodefinovat predikát `keysort`. Jeho definici necháme na procvičení laskavému čtenáři. Sdělíme mu jen pro úplnost, že `keysort(KeyList,SortedKeyList)` setřídí seznam termů tvaru `[Key-cokoliv,...]` vzestupně podle klíče `Key`.

Tento zdrojový text pracuje na základě požadavků 1 a 2 metody LEX. Požadavek 3 si uvedeme dále pouze v principu.

Nechť máme pravidla:

```
rule t1:[flies(X),lays_eggs(X)] ==> [assert(bird(X))].
rule t2:[mammal(X),long_ears(X),eats_carrots(X)] ==> [assert(animal(X,rabbit))].
```

a nechť máme v pracovní paměti:

```
fact(flies(lara),9).
fact(flies(zach),6).
fact(lays_eggs(lara),7).
fact(lays_eggs(zach),8).
fact(mammal(bonbon),3).
fact(long_ears(bonbon),4).
fact(eats_carrots(bonbon),5).
```

Máme tedy:

- dvě instance `rule t1`: `[9,7]` a `[6,8]`
- jednu instanci `rule t2`: `[3,4,5]`

Po setřídění jednotlivých instancí dostaneme `[9,7]`, `[8,6]` a `[5,4,3]`, přičemž `9`, `8` a `5` jsou klíče k instancím. Pak LEX metoda provede toto uspořádání konfliktní množiny:

$$[9,7] > [8,6] > [5,4,3]$$

Tuto množinu se nám však podařilo uspořádat jen podle klíčů. To by se nám však nemuselo podařit, kdybychom například měli v konfliktní množině dvě instance: `[9,7]` a `[9,7,3]`, které mají obě dvě klíč `9`. Proto zde musí přijít na řadu „specifikovanost" z bodu 3 principů metody LEX. Ta rozhodne ve prospěch instance `[9,7,3]`, protože má více specifikovaných předpokladů než instance `[9,7]`:

$$[9,7] < [9,7,3]$$

6.4.3 MEA metoda

MEA metoda je identická s metodou LEX, jen přidává jakýsi filtr navíc. Uveďme si tedy implementaci této metody:

```
select_rule(R,CS) :- refract(CS,CS1),mea_filter(0,CS1,[],CSR),
     lex_sort(CSR,[R|_]).
mea_filter(_,X,_,X) :- not strategy(mea),!.
mea_filter(_,[],X,X).
mea_filter(Max,[r([A/T|Z],B,C,D)|X],Temp,ML) :- T < Max,!,
     mea_filter(Max,X,Temp,ML).
mea_filter(Max,[r([A/T|Z],B,C,D)|X],Temp,ML) :- T = Max,!,
     mea_filter(Max,X,[r([A/T|Z],B,C,D)|Temp],ML).
mea_filter(Max,[r([A/T|Z],B,C,D)|X],Temp,ML) :- T > Max,!,
     mea_filter(T,X,[r([A/T|Z],B,C,D)],ML).
```

`mea_filter` vybere pouze ta pravidla, jejichž instance prvního termu mají maximální možný čas.

6.5 Rámce

6.5.1 Co jsou to rámce

Rámce jsou datovou strukturou, kterou si můžeme představit jako tabulky např.:

savci	value	default	when_updated
kůže	`kožešina`		
rození	`živé`		
nohy		4	`leg_check`

králík	value	default	when_updated
a_kind_of	`savci`		
uši		`dlouhé`	
pohyb	`skoky`		

Rámce (*frames*) si můžeme na této intuitivní úrovni představit jako tyto tabulky, jejichž řádky budeme nazývat sloty (*slots*) a sloupce budeme nazývat facety (*facets*).

Do prologovské programové databáze budeme rámce ukládat pomocí predikátů:

- `frame/2` s parametry:
    1. jméno rámce (např. `savci`)
    2. seznam slotů
- `slot/2` s parametry:
    1. jméno slotu (např. `kůže`)
    2. seznam facetů
- `facet/2` s parametry:
    1. jméno facetu (např. `value`), které bude reprezentováno operátorem
    2. hodnota facetu

Například:

```
frame(name,[slotname1 - [facet11/val11,facet12/val12,...],
            slotname2 - [facet21/val21,facet22/val22,...],
            ...]).
```

Facety budeme dělit do těchto typů:

- `val` – jednoduchá hodnota facetu
- `def` – default
- `calc` – predikát volající výpočet hodnoty slotu
- `add` – predikát, který je volán, když je hodnota přidávána do slotu
- `del` – predikát, který je volán, když je hodnota vyřazována ze slotu

Příklad 6.2

```
frame(man,[ako - [val person],hair - [def short,del bald],weight - [calc male_weight]]).
frame(woman,[ako - [val person],hair - [def long],weight - [calc female_weight]]).
```

Facet `ako` (jehož sémantika je „a kind of") zde funguje jako odkaz na nějaký rámec. Toto je jistý objektový rys rámců, který pomocí tohoto facetu zavádí do rámců dědičnost, kdy rámec `woman` dědí hodnoty všech slotů rámce `person`, které rozšiřuje nebo aktualizuje.

6.5.2 Prohlížení rámců

V tomto odstavci si nadefinujeme predikát `get_frame`, který podle požadavků `ReqList` najde rámec `Thing`.

```
get_frame(Thing,ReqList) :- frame(Thing,SlotList),slot_vals(Thing,ReqList,SlotList).
slot_vals(_,[],_).
slot_vals(T,[Req|Rest],SlotList) :- prep_req(Req,req(T,S,F,V)),
     find_slot(req(T,S,F,V),SlotList),!,slot_vals(T,Rest,SlotList).
slot_vals(T,Req,SlotList) :- prep_req(Req,req(T,S,F,V)),
     find_slot(req(T,S,F,V),SlotList).
prep_req(Slot-X,req(T,Slot,val,X)) :- var(X),!.
prep_req(Slot-X,req(T,Slot,Facet,Val)) :- nonvar(X),X =.. [Facet|Val],
     facet_list(FL),member(Facet,FL),!.
prep_req(Slot-X,req(T,Slot,val,X)).
facet_list([val,def,calc,add,del,edit]).
find_slot(req(T,S,F,V),SlotList) :- nonvar(V),find_slot(req(T,S,F,Val),SlotList),!,
     (Val == V;member(V,Val)).
find_slot(req(T,S,F,V),SlotList) :- member(S-FacetList,SlotList),!,
     facet_val(req(T,S,F,V),FacetList).
find_slot(req(T,S,F,V),SlotList) :- member(ako-FacetList,SlotList),
     facet_val(req(T,ako,val,Ako),FacetList),(member(X,Ako);X = Ako),
     frame(X,HigherSlots),find_slot(req(T,S,F,V),HigherSlots),!.
find_slot(Req,_) :- error(['Frame error looking for: ',Req]).
facet_val(req(T,S,F,V),FacetList) :- FV =.. [F,V],member(FV,FacetList),!.
facet_val(req(T,S,val,V),FacetList) :- member(val ValList,FacetList),
     member(V,ValList),!.
facet_val(req(T,S,val,V),FacetList) :- member(def V,FacetList),!.
facet_val(req(T,S,val,V),FacetList) :- member(calc Pred,FacetList),
     Pred =.. [Functor|Args],CalcPred =.. [Functor,req(T,S,Val,V)|Args],
     call(CalcPred).
```

Pomocí predikátu `req/4` budeme reprezentovat požadavky. Jako parametry budou jméno rámce, požadovaný slot, požadovaný facet a požadovaná hodnota.

Ještě poznamenejme, že operátor `=..` (univ) uspěje například v tomto případě:

```
fact(0,N,F) =.. [fact,0,N,F]
```

A závěrem příklad výpočetního predikátu:

```
female_weight(req(T,S,F,V)) :- get_frame(T,[height-H]),V is H*2.
```

6.5.3 Přidávání rámců

V tomto odstavci si uvedeme implementaci predikátu `add_frame`, který bude zařazovat nové rámce do pracovní paměti.

```
add_frame(Thing,UList) :- old_slots(Thing,SlotList),
     add_slots(Thing,UList,SlotList,NewList),retract(frame(Thing,_)),
     asserta(frame(Thing,NewList)),!.
old_slots(Thing,SlotList) :- frame(Thing,SlotList),!.
old_slots(Thing,[]) :- asserta(frame(Thing,[])).
add_slots(_,[],X,X).
add_slots(T,[U|Rest],SlotList,NewList) :- prep_req(U,req(T,S,F,V)),
     add_slot(req(T,S,F,V),SlotList,Z),add_slots(T,Rest,Z,NewList).
add_slots(T,X,SlotList,NewList) :- prep_req(X,req(T,S,F,V)),
     add_slot(req(T,S,F,V),SlotList,NewList).
add_slot(req(T,S,F,V),SlotList,[S-FL2|SL2]) :- delete(S-FacetList,SlotList,SL2),
     add_facet(req(T,S,F,V),FacetList,FL2).
```

```
add_facet(req(T,S,F,V),FacetList,[FNew|FL2]) :- FX =.. [F,OldVal],
     delete(FX,FacetList,FL2),add_newval(OldVal,V,NewVal),!,
     check_add_demons(req(T,S,F,V),FacetList),FNew =.. [F,NewVal].
add_newval(X,Val,Val) :- var(X),!.
add_newval(OldList,ValList,NewList) :- list(OldList),list(ValList),
     append(ValList,OldList,NewList),!.
add_newval([H|T],Val,[Val,H|T]).
add_newval(Val,[H|T],[Val,H|T]).
add_newval(_,Val,Val).
check_add_demons(req(T,S,F,V),FacetList) :- get_frame(T,S-add(Add)),!,
     Add =.. [Functor|Args],AddFunc =.. [Functor|req(T,S,F,V)|Args],call(AddFunc).
check_add_demons(_,_).
delete(X,[],[]).
delete(X,[X|Y],Y) :- !.
delete(X,[Y|Z],[Y|W]) :- delete(X,Z,W).
```

Predikát `del_frame`, který bude odstraňovat rámce z pracovní paměti bude analogický, s inverzním účinkem. Jeho případnou implementaci necháváme na laskavém čtenáři.

6.5.4 Příklad znalostní báze v rámcích

```
frame(tubenose,[level-[val order],nostrils-[val external_tubular],
     live-[val at_sea],bill-[val hooked]]).
frame(albatross,[ako-[val tubenose],level-[val family],size-[val large],
     wings-[val long_narrow]]).
frame(legsan_albatross,[ako-[val albatross],level-[val species],color-[val white]]).
frame(black_footed_albatross,[ako-[val albatross],level-[val species],
     color-[val dark]]).
```

Laskavému čtenáři necháváme na závěr k promyšlení, jaké hodnoty proměnné `X` vrátí dotazy:

- `?- get_frame(X,[color - dark,wings - long_narrow]).`
- `?- get_frame(X,[wings - long_narrow]).`
- `?- get_frame(X,[wings - long_narrow,level - L]).`

Literatura

[1] David Krásenský, *Logické programování*, ZKUSTO 1994