Prohledávání stavového prostoru

Prohledávání grafu - potřebné prostředky Prologu

- **stromy a acyklické grafy** - stačí čistý Prolog
- **konečné grafy s případným cyklem** - Prolog + negace *
- **nekonečné grafy** (lokálně konečné) -
  Prolog + negace* + predikáty 2. řádu **

* nutná pro kontrolu cyklu

** prohledávání do šířky

Predikáty měnící databázi programu assert(+*Clause*), retract (+*Clause*)

assert/1	přidává obsah svého argumentu **ZA** definici predikátu téhož jména
asserta/1	přidává obsah arg. **PŘED** tuto definici
retract (+*Clause*)	najde **první** klauzuli, kterou lze unifikovat s ***Clause***, a **zruší ji**

POZOR!

Predikát v hlavě *Clause* musí být deklarován jako DYNAMICKÝ, tedy takový, že jeho definice se v průběhu výpočtu může měnit.

Například

:- dynamic fronta/1.

Upozorňuje na to, že predikát **fronta** se v průběhu programu může měnit!

Použití predikátů assert(+*Clause*), retract (+*Clause*)

Obdobu predikátu **bagof(+X,+Cíl(X,_),-Seznam)** můžeme naprogramovat právě pomocí **assert** a **retract**:

Náš vlastní predikát nazvěme **findall_p.** Musíme použít pomocné konstr.:

% * konstantu **konec** jako značku, že další možnosti nejsou a

% * predikát **fronta** pro uložení nalezených instancí za **X** v dotazu **Cil**

```
:- dynamic fronta/1.
findall_p(X, Cil, XSeznam) :- call (Cil), assert (fronta(X) ), fail.
findall_p(X, Cil, XSeznam) :-   assert (fronta(konec) ),
                                shrn_fronta ( XSeznam ).
shrn_fronta ( Sezn ) :- retract (fronta (X) ), !,
    ( X == konec,!,Sezn = [] ; Sezn = [X| Z],shrn_fronta ( Z ) ).
```



```
:- dynamic fronta/1.
hr(a,b).  hr(a,c).  hr(a,d).  hr(b,e).  hr(b,f).  hr(c,g). hr(c,h).  hr(c,a).
hr(g,k).  hr(g,l).  hr(d,i).  hr(d,j).  hr(j,m). hr(j,n).
cil(f).   cil(i).   cil(l).   cil(n).

findall_p(X,Cil,XSeznam) :- call(Cil), assert(fronta(X)), fail.
findall_p(X,Cil,XSeznam) :- assert(fronta(konec)),shrn_fronta(XSeznam ).
shrn_fronta(Sezn) :- retract(fronta(X)),!,
    (X == konec,!,Sezn = []; Sezn = [X| Z],shrn_fronta(Z)).
```

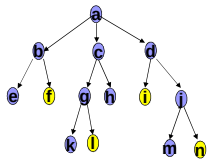
?- findall_p(X, hr(a, X), L). X = X L = [b, c, d] Yes (0.00s cpu)

?- findall_p(X, hr(g, X), L). X = X L = [k, l] Yes (0.00s cpu)

Prohledávání do šířky s rekonstrukcí cesty **bf_path**

```
:- dynamic edge/2.
% c_sir1(+VychoziStav,-CilStav,-Cesta)
c_sir1(X,X,[X]):-cil(X).
c_sir1(X,Y,T):-vlna1([X],Y),cil(Y),cesta(X,Y,T).
vlna1([],konec):-nl,write('dalsi cilove uzly nejsou'),!.
vlna1([H|T],H):-cil(H).
vlna1([H|T],Z):-pridej_za1(T,H,N),vlna1(N,Z).
pridej_za1(L,H,N):-primi_naslednici(H,S),append(L,S,N),pamatuj(H,S).
primi_naslednici(H,S):-bagof(X,hr(H,X),S),!.
primi_naslednici(H,[]).
pamatuj(Z,[]).
% pamatuj (+Stav,+Seznam_stavu) zapamatovává si info o všech hranách vedoucích ze Stav do Seznam_stavu
pamatuj(Z,[H|T]):-asserta(edge(Z,H)),pamatuj(Z,T).
% cesta(+VychStav,+CilStav,-Cesta) se konstruuje „zpětně" : proto se čte ←
cesta(X,X,[X]):-!.      cesta(X,Y,[Y|T]):-edge(H,Y),cesta(X,H,T).
```

Jak hledá program **bf_path** cestu do cílových uzlů tohoto grafu?



?- c_sir1(a, X, L).

X = f L = [f, b, a] Yes (0.00s cpu, solution 1, maybe more)

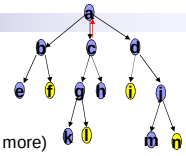
X = i L = [i, d, a] Yes (0.00s cpu, solution 2, maybe more)

X = l L = [l, g, c, a] Yes (0.00s cpu, solution 3, maybe more)

X = n L = [n, j, d, a] Yes (0.00s cpu, solution 4, maybe more)

No (0.00s cpu)

Jak hledá program **bf_path** cestu do cílových uzlů zde?



?- c_sir1(a, X, L).

X = f L = [f, b, a] Yes (0.00s cpu, solution 1, maybe more)

X = i L = [i, d, a] Yes (0.00s cpu, solution 2, maybe more)

X = i L = [i, d, a] Yes (0.02s cpu, solution 3, maybe more)

X = l L = [l, g, c, a] Yes (0.02s cpu, solution 4, maybe more)

X = l L = [l, g, c, a] Yes (0.02s cpu, solution 5, maybe more)

X = n L = [n, j, d, a] Yes (0.03s cpu, solution 6, maybe more)

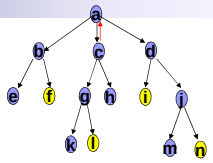
X = n L = [n, j, d, a] Yes (0.03s cpu, solution 7, **maybe more**)

Opakování je způsobeno existencí smyčky. To lze ošetřit tím, že si ještě **zapamatujeme uzly, které už byly analyzovány** (rozvinuty) a ty do „agendy pro zpracování" nedoplňujeme!

Varianta bf_path s kontrolou smyček bf_loop_check

```
:- dynamic edge/2.                    :- dynamic analyzed_node/1.
c_sir2(X,X,[X]):-cil(X).
c_sir2(X,Y,T):-vlna2([X],Y),cil(Y),cesta(X,Y,T).
    vlna2([],konec):-nl,write('dalsi cilove uzly nejsou'),!.
    vlna2([H|T],H):-cil(H).
    vlna2([H|T],Z):-pridej_za2(T,H,N),vlna2(N,Z).
pridej_za2(L,H,N):-assert(analyzed_node(H)) ,
    primi_naslednici(H,S),vynech(S,S1),
    append(L,S1,N),pamatuj(H,S).
vynech([],[]).
vynech([H|T],T1):-analyzed_node(H),!,vynech(T,T1).
vynech([H|T],[H|T1]):-vynech(T,T1).
```


Jak hledá program **bf_loop-check** cestu do cílových uzlů pro tenýž graf?



?- c_sir2(a, Y, T).

Y = f T = [f, b, a] Yes (0.00s cpu, solution 1, maybe more)

Y = i T = [i, d, a] Yes (0.02s cpu, solution 2, maybe more)

Y = l T = [l, g, c, a] Yes (0.03s cpu, solution 3, maybe more)

Y = n T = [n, j, d, a] Yes (0.03s cpu, solution 4, maybe more)

No (0.05s cpu)

Prohledávání stavového prostoru

- **Stavový prostor** = implicitně definovaný graf, ve kterém hrany odpovídají povoleným krokům. Popis úlohy prostřednictvím predikátů:
  - **move(Stav, Aplikovatelná_akce)**, který nabízí všechny akce aplikovatelné v daném stavu
  - **update(Stav, Aplikovatelná_akce, Výsledný_stav_po_akci)**, který generuje stav, který je výsledkem aplikace zvolené akce
  - **init_state**(S), **final_state**(S)
- Postup řešení podobný jako u prohledávání grafu.

Princip řešení úlohy k-v-z2

Reprezentace stavů

wgc(Pozice_lodky, Seznam_obj_vlevo, Seznam_obj_vpravo)

Reprezentace akcí: akce charakterizována nákladem loďky

```
% Popis výchozího a cílového stavu
init_state(wgc, wgc(left,[w,g,c],[])).
final_state(wgc, wgc(right,[],[w,g,c])).
% Popis podmínek pro vykonání akce v daném stavu
move(wgc(left,L,R),Cargo):- element(Cargo,L).
move(wgc(right,L,R),Cargo):- element(Cargo,R).
move(wgc(B,L,R),alone).
```

```
% Výsledný stav po vykonání akce
update(wgc(B,L,R),Cargo,wgc(B1,L1,R1)):-
    update_boat(B,B1), update_banks(Cargo,B,L,R,Li1,Ri1),
    o(Li1,L1),o(Ri1,R1).
% ukolem predikatu o(+X,-Z) je usporadat seznam „sestupne"
update_boat(left,right).        update_boat(right,left).

% update_banks(Cargo,
   BoatInitPosition,LeftBankInit,RightBankInit,LFinal,RFinal).
update_banks(alone,B,L,R,L,R).
update_banks(Cargo,left,L,R,L1,R1)   :- delete(Cargo,L,L1),
    insert(Cargo,R,R1),!.
update_banks(Cargo,right,L,R,L1,R1) :- ...
```



```
% Kontrola nebezpečí hrozícího po vykonání akce
legal(wgc(left,L,R)):- not illegal(R).
legal(wgc(right,L,R)):- not illegal(L).
illegal([w,g]).
illegal([g,c]).
% Pomocné predikáty, napr. „normální tvar" ignorující uspořádání
o([],[]).          o([X],[X]).
o([w,g],[w,g]).    o([g,w],[w,g]).
o([w,c],[w,c]).    o([c,w],[w,c]).
o([g,c],[g,c]).    o([c,g],[g,c]).
o(P,[w,g,c]) :- l(P,3).
% Další pomocné predikáty
% m: "member", d:"delete", i:"insert, l:"lenth"
```


Obecný program pro prohledávání stavového prostoru

```
% Alg. Generující a prohledávající stav.prostor od vých stavu initial_state
% k cílovému final_state využívající následující popis světa:
% move(Stav, Aplikovatelná_akce)
% update(Stav, Aplikovatelná_akce, Výsledný_stav_po_akci)

solve_dfs(State,History,[]):-final_state(State).
solve_dfs(State,History,[Move|Moves]):-
      move(State,Move),
      update(State,Move,State1), legal(State1),
      not member(State1,History),
      solve_dfs(State1,[State1|History],Moves).
% Testování na konkrétní úloze
test_dfs(Problem,Moves):-
    initial_state(Problem,State), solve_dfs(State,[State],Moves).
```


?- test_dfs(wgc, M).

- M = [g, alone, w, g, c, alone, g], Yes (0.00s cpu, solution 1, maybe more)
- M = [g, alone, c, g, w, alone, g], Yes (0.02s cpu, solution 2, maybe more)
- No (0.02s cpu

Řešení úlohy: proveďte akce v **M** postupně zleva doprava, tj. převez kozu → vrať se → převez vlka → převez kozu → převez zelí → vrať se → převez kozu

Poznámky k právě navrženému řešení úlohy k-v-z2

- **Program v této podobě řešení nalezne!**
- **Pokud se tak v jiném případě nestane, je vhodné**
  - Pro odhalení důvodů je rozumné přidat tisk do solve_dfs, aby bylo zřejmé, kudy pokusy o řešení běží.
  - **Ujistit se, zda program neopakuje mnohokrát zbytečné akce**. Je vhodné zajistit, aby se jako další akce nevybrala ta, která právě skončila. Dál je možné *přidat parametr* se jménem poslední akce a to kontrolovat, …
  - Díky tisku se může ukázat, že **popis nelegálních stavů** ani popis cílového stavu nejsou úplné (permutace).
  - Použít **bfs**