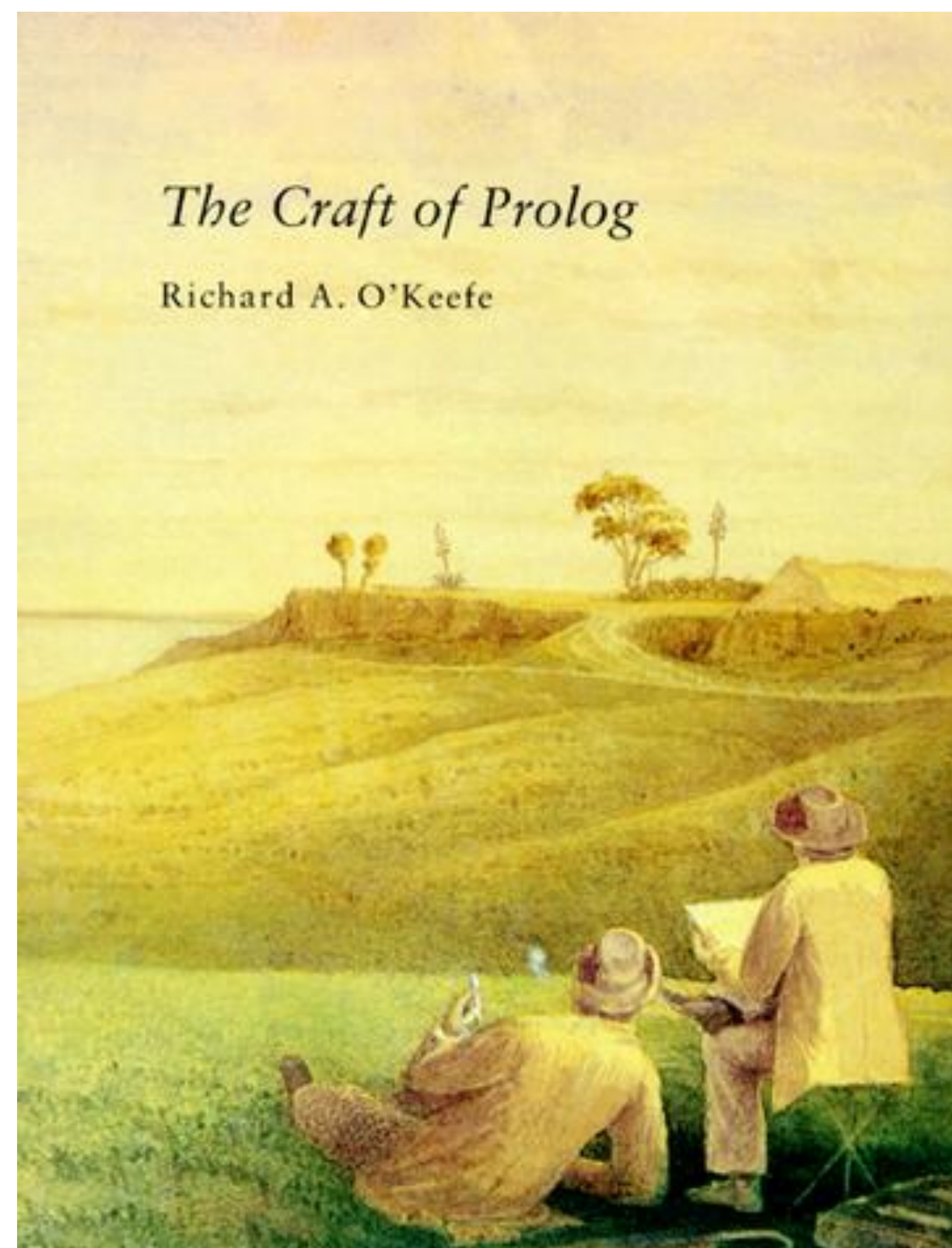







# Neprocedurální programování

## Prolog 5


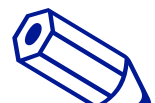


### Prohledávání stavového prostoru



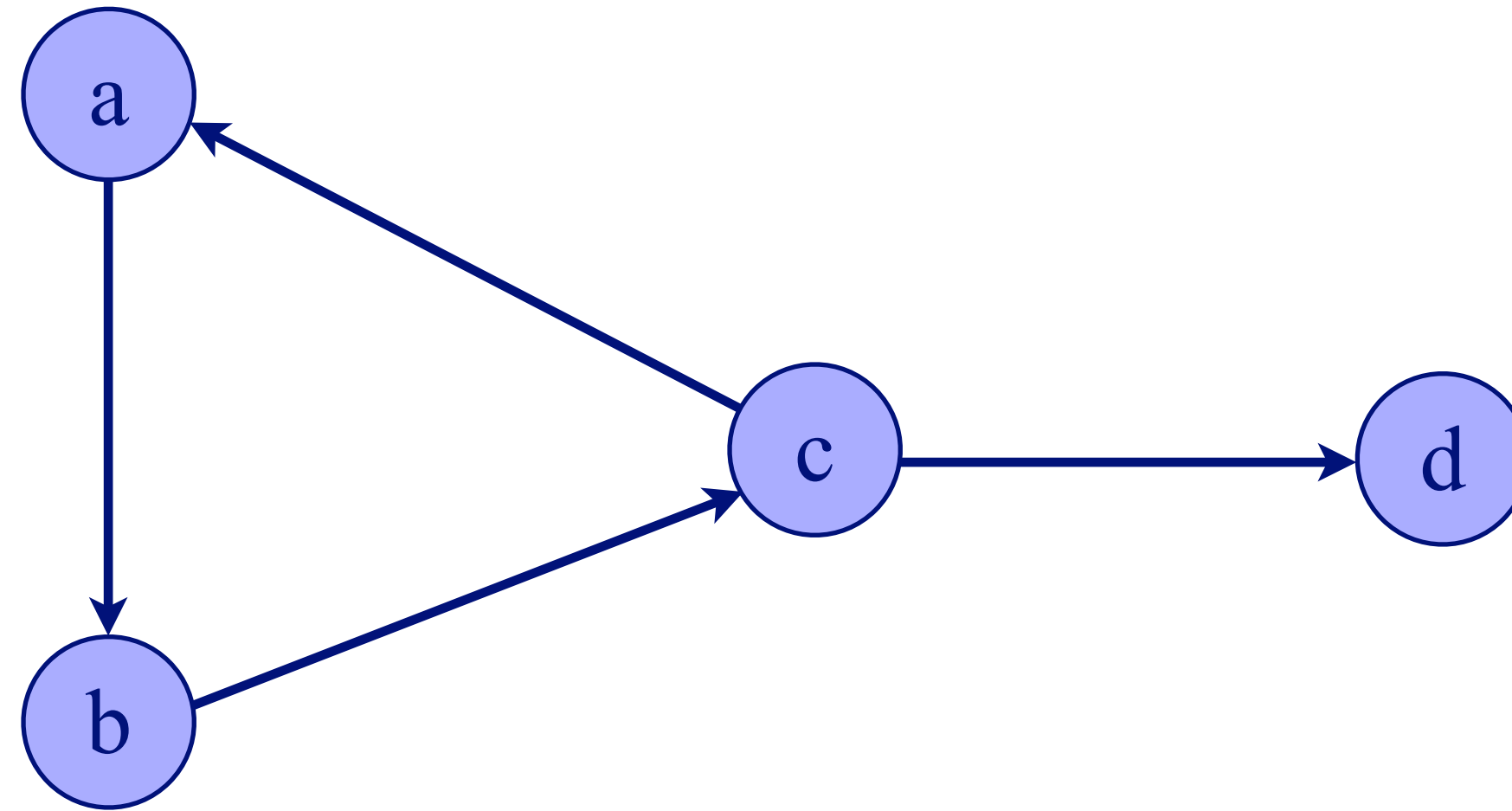
# Co bylo minule

-  Seznamy seznamů
-  Neúplně definované datové struktury
-  Řez a negace
-  Vestavěné predikáty – test struktury termu
-  Symbolická manipulace s výrazy

# Osnova

-  Prohledávání grafů (DFS, IDS, BFS)
-  Prohledávání stavového prostoru
  - příklad: farmář, vlk, koza, zelí
-  Shromáždění všech výsledků dotazu
-  Vstup a výstup
  - příklad : Eliza

# Grafové algoritmy



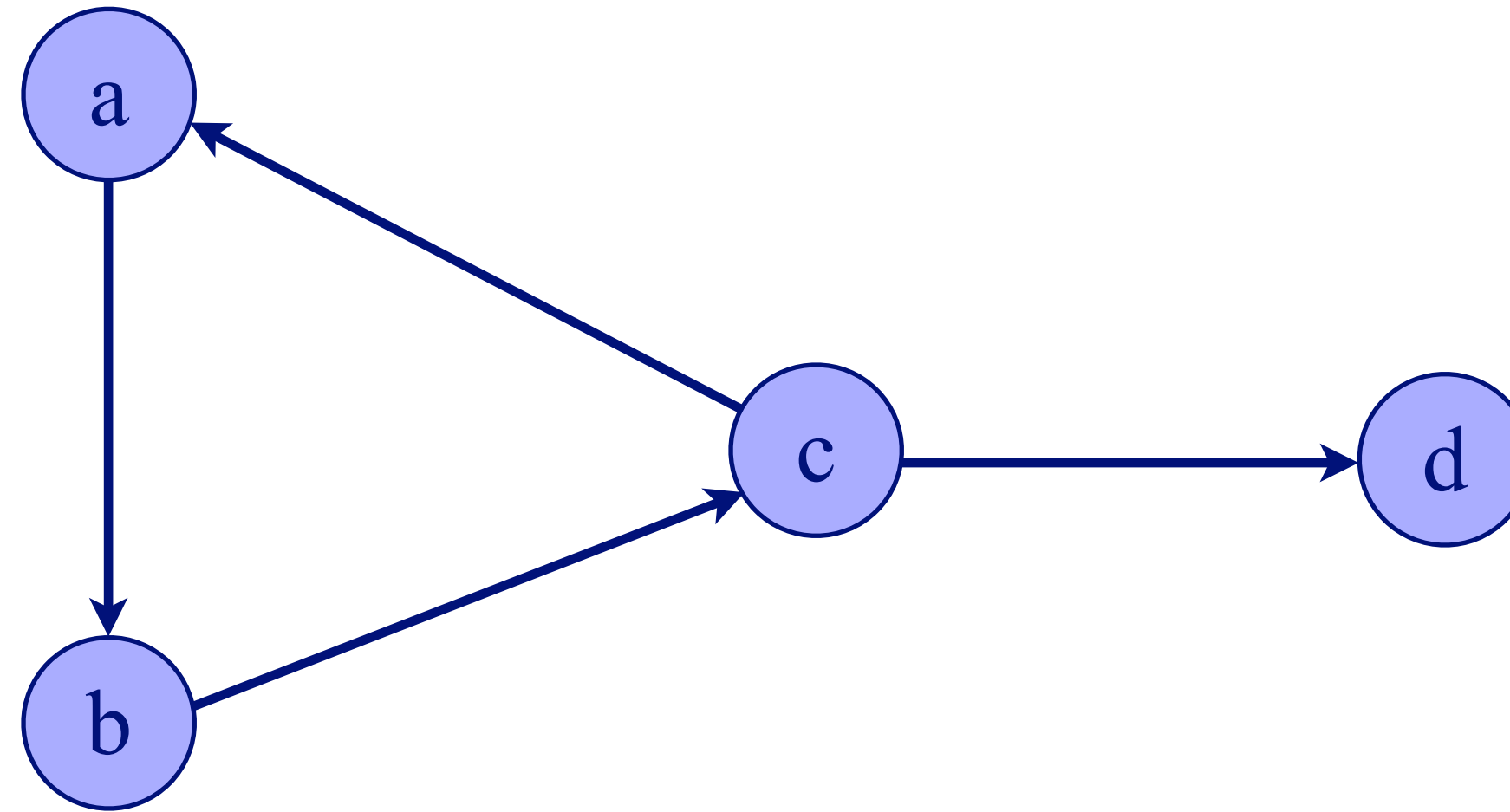
## Reprezentace grafu

`% fakty`

`vrchol(a). vrchol(b). vrchol(c). vrchol(d).`

`hrana(a,b). hrana(b,c). hrana(c,a). hrana(c,d).`

# Grafové algoritmy



## Reprezentace grafu

% složenými termy

- `graf([a,b,c,d], [h(a,b),h(b,c),h(c,a),h(c,d)])`
- `[a->[b],b->[c],c->[a,d],d->[]]`

# Grafy: rozhraní

`vrchol(?Vrchol, +Graf)`

`vrchol(V, G) :- member(V->_, G).`

`hrana(?Vrchol1, ?Vrchol2, +Graf)`

`hrana(V1, V2, G) :- member(V1->Sousedce, G),  
member(V2, Sousedce).`

`sousedce(?Vrchol, ?Sousedce, +Graf)`

`soused(V, S, G) :- member(V->S, G).`

Dále jen

- `hrana(Vrchol1, Vrchol2)`
- `sousedce(Vrchol, Sousedce)`



# Grafy: dosažitelnost

Hledání cesty v grafu průchodem do hloubky (Depth First Search)

```
% dfs(+Start,?Cil):- existuje cesta z vrcholu  
%                               Start do vrcholu Cil?
```

```
dfs(X,X).
```

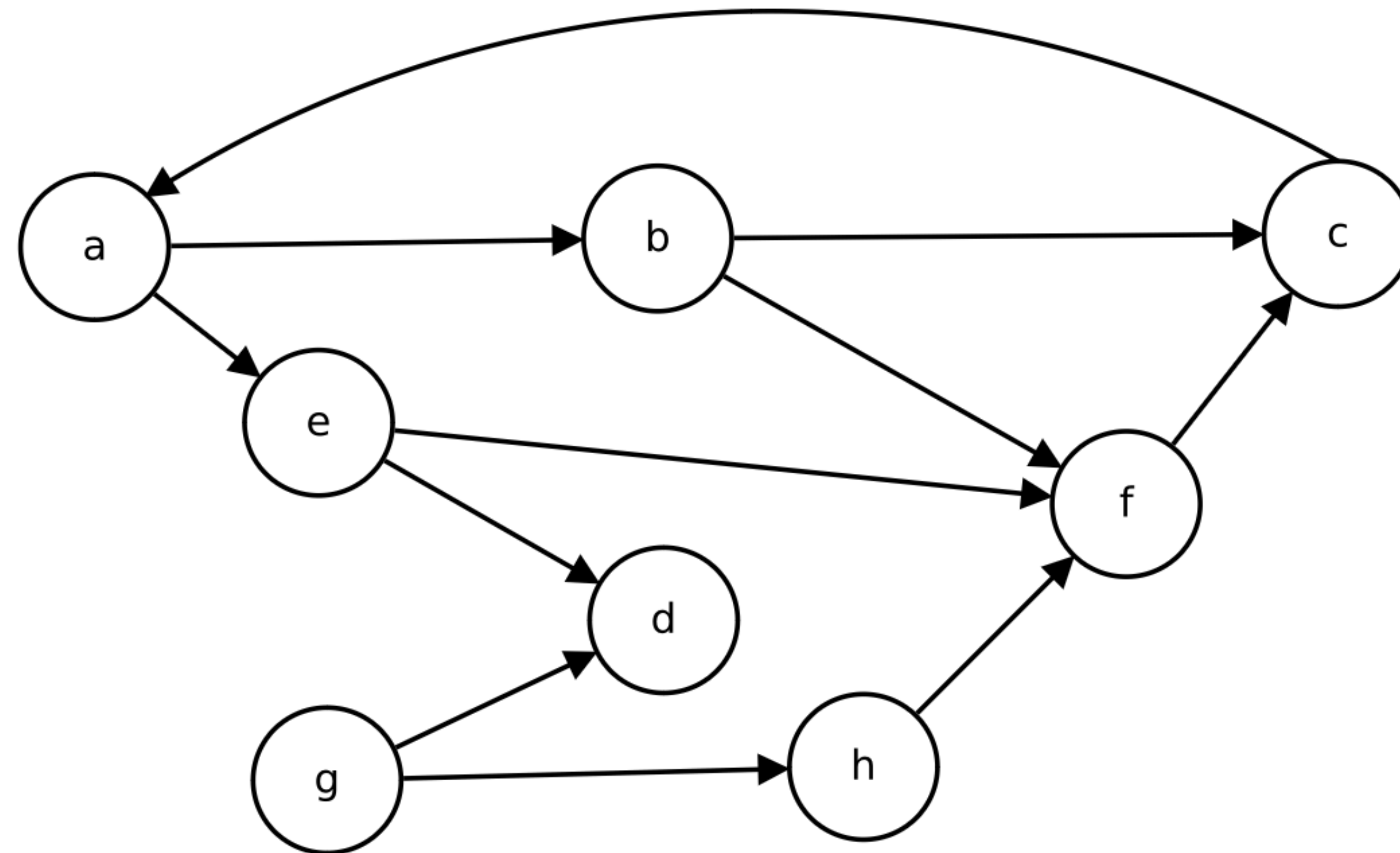
```
dfs(X,Z):- hrana(X,Y), dfs(Y,Z).
```

✗ Korektní jen pro **acyklické** grafy!

# Grafy: příklad

% příklad orientovaného grafu

[ a -> [b, e], b -> [c, f], c -> [a], d -> [], e -> [d, f],  
f -> [c], g -> [d, h], h -> [f] ].





# Grafy: průchod do hloubky

```
dfs(X,Y):- dfs(X,Y,[X]).
```

```
% dfs(X,Y,Nav) :- Nav je seznam již navštívených  
%                      vrcholů.
```

```
dfs(X,X,_).
```

```
dfs(X,Z,Nav):- hrana(X,Y), \+ member(Y,Nav),  
                dfs(Y,Z,[Y|Nav]).
```

👉 **Problém:** `dfs/3` nevrací nalezenou cestu !

`maplist(\==(Y),Nav)`

# Grafy: průchod do hloubky

Predikát, který vrátí i nalezenou cestu

```
% dfs(+X,?Y,?Cesta):- Cesta je seznam vrcholů  
%                       na cestě z X do Y.
```

```
dfs(X,Y,Cesta):- dfs(X,Y,[X],Cesta).
```

```
dfs(X,X,_,[X]).
```

```
dfs(X,Z,Nav,[X|Ys]):- hrana(X,Y), \+ member(Y,Nav),  
                      dfs(Y,Z,[Y|Nav],Ys).
```

# Grafy: iterativní prohlubování

DFS do zadané maximální hloubky

```
% ids(+X,?Y,+MaxDelka,?Cesta):-  
%           hledá Cestu z X do Y  
%           zadané maximální délky Maxdelka  
%           iterativním prohlubováním.  
  
ids(X,Y,MaxDelka,Cesta):-  
    MaxPocetVrcholu is MaxDelka+1,  
    between(1,MaxPocetVrcholu,N),  
    length(Cesta,N), dfs(X,Y,Cesta).
```

# Grafy: průchod do šířky

Hledání cesty v grafu **průchodem do šířky** (Breadth First Search)

- použití fronty již nalezených cest
- která reprezentuje BFS-strom

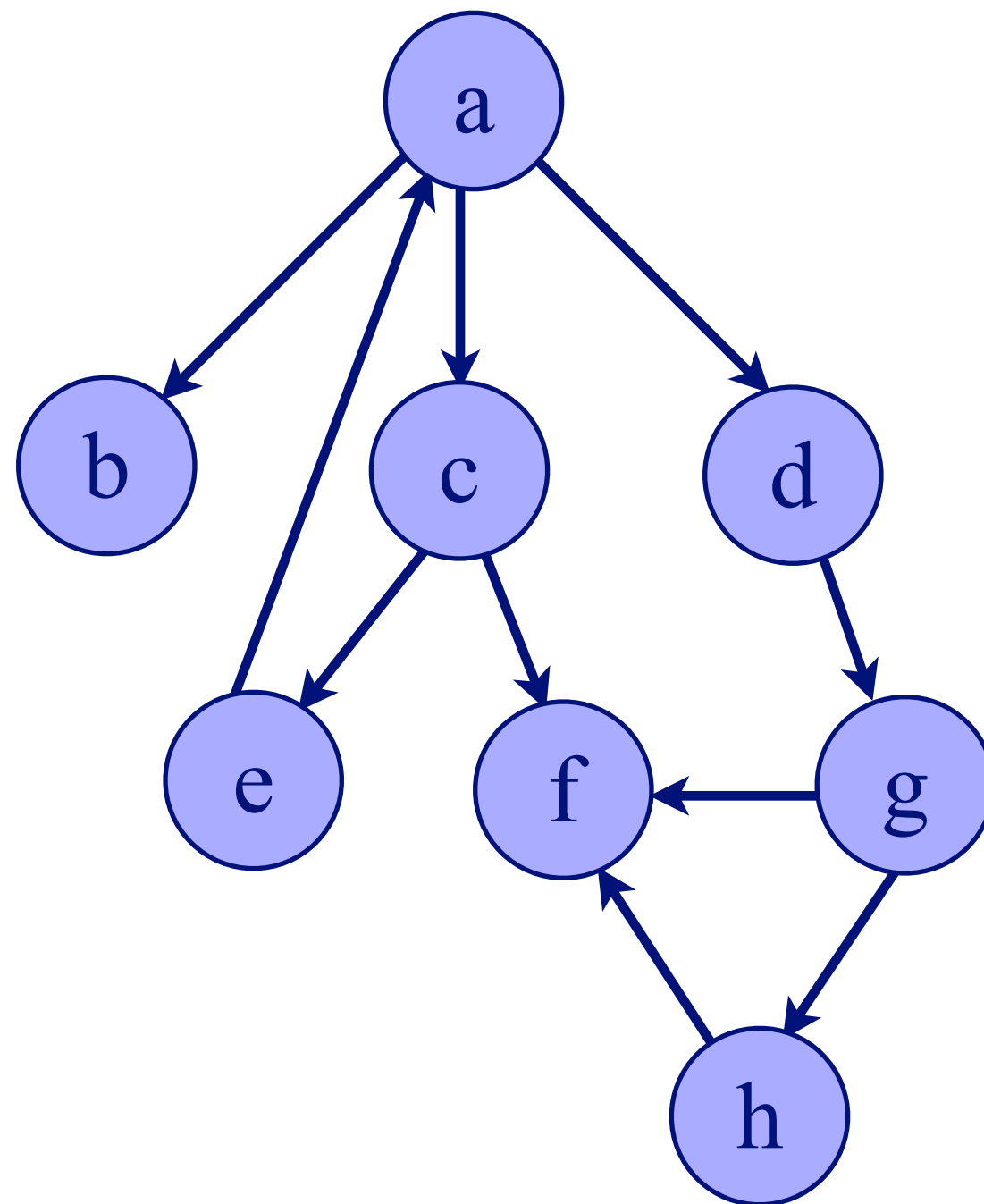
```
% bfs(+Start,+Cil,-Cesta):- Cesta z vrcholu Start  
%                             do vrcholu Cil nalezená  
%                             průchodem do šířky.
```

```
bfs(Start,Cil,Cesta):- bfs1([[Start]],Cil,Cesta).
```



fronta cest

# ☀ Příklad – fronta cest



? – `bfs(a, f, Cesta)` .

[ [a] ]

[ [b,a], [c,a], [d,a] ]

[ [c,a], [d,a] ]

[ [d,a], [e,c,a], [f,c,a] ]

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

[ [f,c,a], [g,d,a] ]

# Grafy: průchod do šířky

```
% pomocny predikat
```

```
vloz_do_hlavy(Xs, X, [X|Xs]).
```

```
% Cesta do Cile nalezena
```

```
% bfs1(Fronta, Cil, Cesta)
```


```
bfs1([Xs|_], Cil, Cesta):- Xs=[Cil|_],  
                           reverse(Xs, Cesta).
```



# Grafy: průchod do šířky

```
% odebere z fronty první cestu  
% najde všechna acyklická prodloužení  
% a vloží je na konec fronty
```

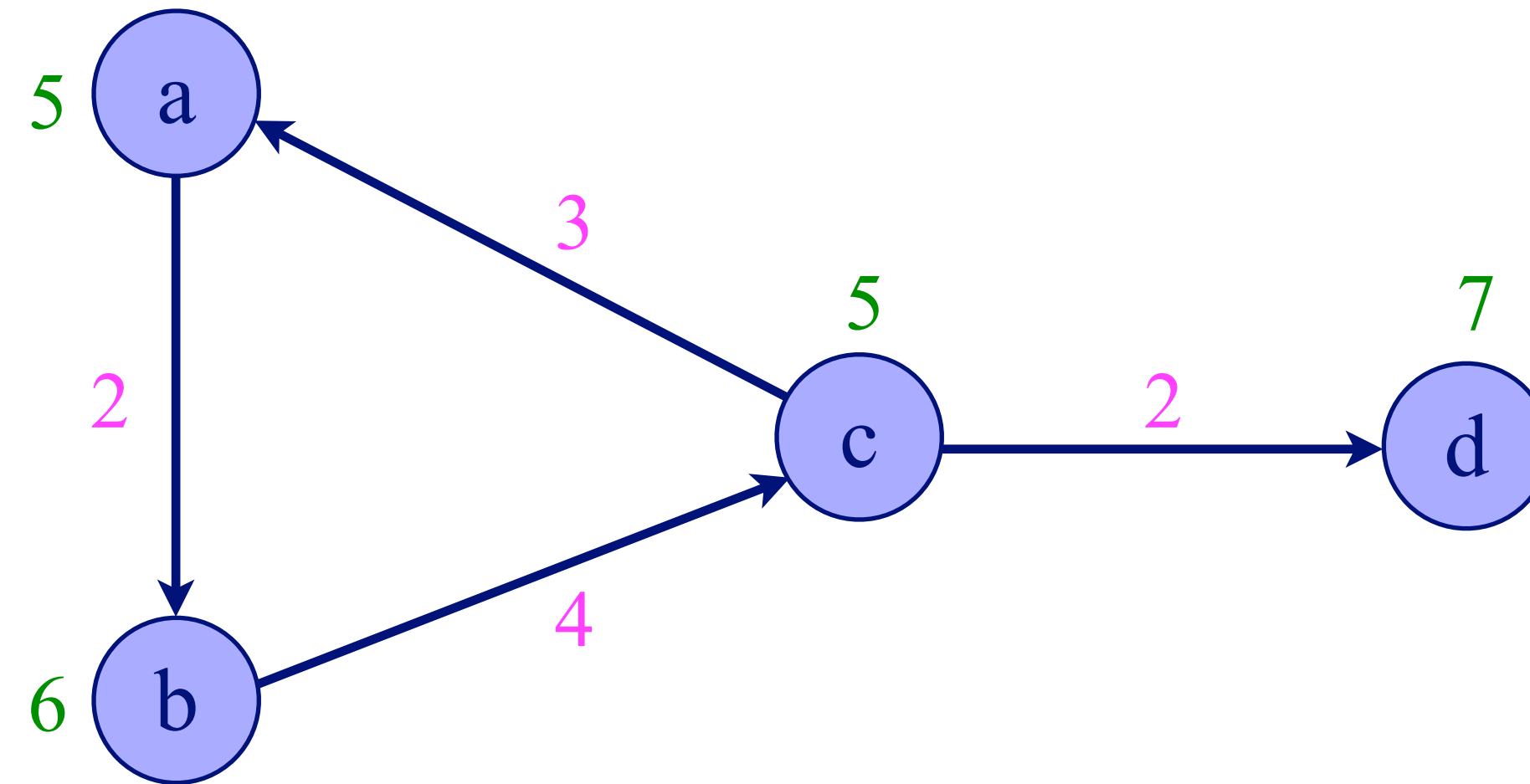
```
bfs1([ [X|Xs] | Xss ], Cil, Cesta) :-  
    sousede(X, Sousede),  
    subtract(Sousede, [X|Xs], NoviSousede),  
    maplist(vloz_do_hlavy([X|Xs]), NoviSousede,  
                                                    NoveCesty),  
    append(Xss, NoveCesty, NovaFronta), !,  
    bfs1(NovaFronta, Cil, Cesta).
```



# Problémy

- ❶ Navrhnete efektivnější verzi predikátu `bfs1/3` s `frontou`
  - realizovanou prostřednictvím `rozdílových seznamů`
  - či „neprocedurálně“ ve tvaru `Predni / Zadni`
- ❷ Implementujte alternativní verzi průchodu do šířky, v níž budeme cesty prodlužovat jen vrcholy dosud `nenavštívenými`.  
Pro reprezentaci množiny všech navštívených vrcholů lze využít asociativní seznam, implementovaný ve standardní knihovně SWI Prologu prostřednictvím vyvážených binárních vyhledávacích stromů [association list].

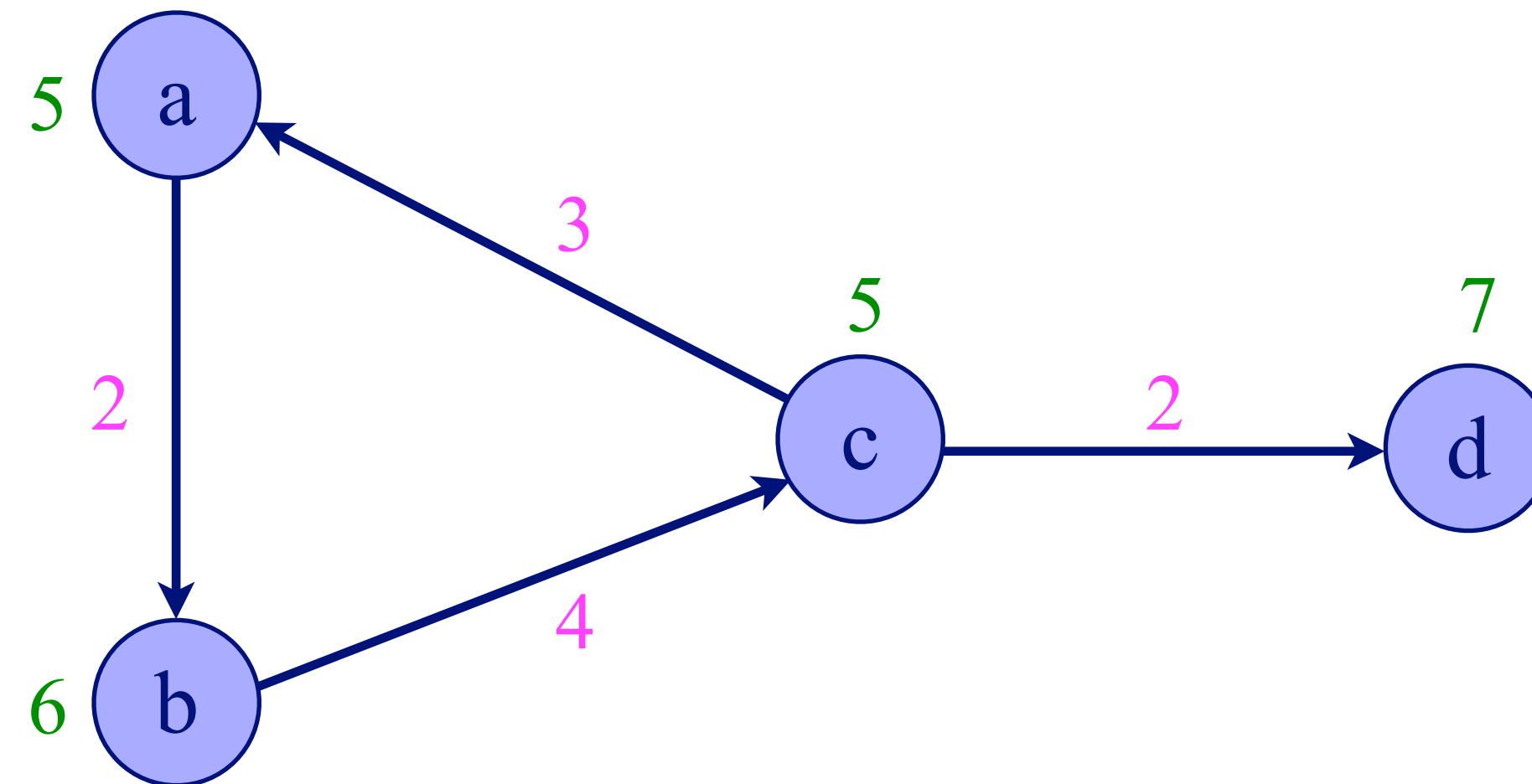
# Grafy s ohodnocením



## Reprezentace grafu

- `graf([a/5,b/6,c/5,d/7],  
[h(a,b,2),h(b,c,4),h(c,a,3),h(c,d,2)])`
- `[a/5->[b/2],b/6->[c/4],c/5->[a/3,d/2],d/7->[]]`

# Grafy s ohodnocením



## Rozhraní

- `vrchol(?Vrchol, ?Ohodnoceni, +Graf)`
- `hrana(?Vrchol1, ?Vrchol2, ?Ohodnoceni, +Graf)`
- `soused(?Vrchol, ?Soused, +Graf)`

# Grafy: problém nejkratší cesty

## Graf bez ohodnocení hran

- délka cesty = # hran
- nejkratší cesta  $\Rightarrow$  **bfs** / 3

## Graf s nezáporným ohodnocením hran

- cena cesty =  $\sum$  ohodnocení hran
- nejkratší cesta  $\Rightarrow$  **dijkstra** / 3

## Reprezentace cesty

- seznam  $[a, b, c] \Rightarrow$  term  $c(Cena, [a, b, c])$
- fronta cest  $\Rightarrow$  prioritní fronta cest (s cenami)
- výběr nejdříve přidané cesty  $\Rightarrow$  výběr cesty minimální cen

# Prohledávání stavového prostoru

**Příklad:** Úloha o farmáři, vlku, koze a zelí

- farmář převáží vlka, kozu a zelí na druhý břeh
- do lodky se vejdou vždy jen dva objekty
- farmář nesmí zanechat na jednom břehu
  - » kozu & zelí
  - » vlka & kozu

Řešení úlohy: posloupnost stavů

Reprezentace stavu

- $s(\text{Farmer}, \text{Vlk}, \text{Kozu}, \text{Zelí})$
- počáteční stav:  $s(1, 1, 1, 1)$
- cílový stav:  $s(p, p, p, p)$



# ☀ Příklad: Farmář, vlk, koza, zelí

```
proti(l,p).      proti(p,l).  
prevoz(s(F,V,K,Z),s(F1,V,K,Z)) :- proti(F,F1).  
prevoz(s(F,F,K,Z),s(F1,F1,K,Z)) :- proti(F,F1).  
prevoz(s(F,V,F,Z),s(F1,V,F1,Z)) :- proti(F,F1).  
prevoz(s(F,V,K,F),s(F1,V,K,F1)) :- proti(F,F1).
```

# Farmář, vlk, koza, zelí

Predikát **bezpecny** / 1

- definuje "bezpečný" stav

**bezpecny** ( s ( F , V , F , Z ) ) .

**bezpecny** ( s ( F , F , K , F ) ) :- proti ( F , K ) .

Predikát **dalsi** / 2 generuje k zadanému stavu všechny následující stavy, které jsou bezpečné

**dalsi** ( Stav1 , Stav2 ) :- prevoz ( Stav1 , Stav2 ) ,  
bezpecny ( Stav2 ) .

# Farmář, vlk, koza, zelí

Zbývá nalézt cestu v grafu

- s vrcholy  $s(F, V, K, Z)$
- a hranami  $hrana(X, Y) :- dalsi(X, Y)$
- z vrcholu  $s(1, 1, 1, 1)$  do vrcholu  $s(p, p, p, p)$

```
fvkz(Resení) :- dfs(s(1,1,1,1), s(p,p,p,p), Resení).
```

```
fvkz(Resení) :- bfs(s(1,1,1,1), s(p,p,p,p), Resení).
```

```
sousedé(X, S) :- findall(Y, dalsi(X, Y), S).
```



# Problém: Útěk před Divokým honem

Zaklínači **Geralt** a **Vesemir**, čarodějka **Triss** a princezna **Ciri** prchají před družinou přízraků zvanou Divoký hon, jejich náskok však činí pouhých **17 minut**. Právě dorazili k rozbouřené řece, kterou lze překonat pouze na Geraltově koni zvaném Klepna, který unese jednoho až dva jezdce.

Víme, že Klepna překoná řeku

- s **Vesemirem** za **10 minut**,
- s **Geraltem** za **5 minut**,
- s **Triss** za **2 minuty**
- a s **Ciri** za **1 minutu**.

Pokud kůň nese dva jezdce, potřebuje k překonání řeky **maximum** ze dvou časů. Dále víme, že Klepna musí mít **alespoň jednoho** jezdce.

Mohou se naši hrdinové dostat na druhý břeh do **17 minut**?

# Shromáždění všech výsledků dotazu

Vestavěné predikáty `bagof/3`, `setof/3`, `findall/3`

`bagof(±Objekt, ±Cil, -SeznamObjektuSplnujicichCil)`

Pokud `Cil` nelze splnit, `bagof` selže

`Seznam` může obsahovat opakované výskyty

Pokud `Cil` obsahuje volnou proměnnou `X`, která není obsažena v `Objektu`

- `bagof` postupně vrátí všechny výsledky
- pro všechny různé hodnoty `X`, pro něž `Cil` uspěje
- `X^Cil` všechna řešení bez ohledu na hodnoty `X`

# ☀ Příklad

```
trida(b,sou). trida(a,sam). trida(c,sou).  
trida(e,sam). trida(d,sou).
```

Dotazy

```
?- bagof(P, trida(P,sou), Pismena).
```

```
    Pismena = [b,c,d]
```

```
?- bagof(P, trida(P,T), Pismena).
```

```
    T = sou, Pismena = [b,c,d] ;
```

```
    T = sam, Pismena = [a,e]
```

```
?- bagof(P, T^trida(P,T), Pismena).
```

```
    Pismena = [b,a,c,e,d]
```



# Vestavěné predikáty: setof

setof / 3

- jako bagof / 3, ale
- vrátí **uspořádaný** seznam
- bez duplicit

?– **setof(T/P, trida(P,T), Pismena).**

Pismena = [sam/a,sam/e,sou/b,sou/c,sou/d]

# Vestavěné predikáty: findall

`findall/3`

- jako `bagof/3`, ale
- shromáždí všechna řešení **bez ohledu na volné proměnné**, nevyskytující se v `Cil`i
- **vždy** uspěje
  - » pokud `Cil` nelze splnit, vrátí `[]`

? – `findall(P, trida(P,T), Pismena).`

`Pismena = [b,a,c,e,d]`

# Vstup a výstup: termy

V/V termů

`read( ?T )` přečte z aktuálního vstupu jeden term  
(ukončený tečkou) a unifikuje s **T**

`write(+T)` vypíše na aktuální výstup hodnotu termu **T**

- s právě platnými hodnotami proměnných v termu **T** obsažených

# Vstup a výstup: znaky

## Znakový vstup

`get_char( ?C )` unifikuje **C** s dalším znakem na vstupu

`get_code( ?C )` unifikuje **C** s kódem dalšího znaku na vstupu

# Vstup a výstup: znaky

## Znakový výstup

`put_char(Z)` vypíše znak **Z** na aktuální výstup

`put_code(C)` vypíše znak s kódem **C** na aktuální výstup

`tab(N)` vypíše **N** mezer

`nl` nový řádek

# Vstup a výstup: proudy

Implicitní vstup - klávesnice, výstup - obrazovka

- atom `user`

Edinburgský model

- `see(+F)` nastaví vstup ze souboru `F`  
» `see('C:\prolog\data.pl')`
- `seen/0` uzavře aktuální vstup, `see(user)`
- `seeing(-F)` dotaz na aktuální vstupní soubor
- `tell/1`, `told/0`, `telling/1` analogicky pro výstup



# Vstup a výstup: cykly

Standardní predikát `repeat/0`

```
repeat.
```

```
repeat :- repeat.
```

☀ Příklad

```
seeing(In),      % zjistí a uschová  
telling(Out),    % aktuální V/V  
see(F1),         % otevře vstupní soubor  
tell(F2),        % otevře výstupní soubor
```

# Vstup a výstup: příklad

```
repeat,                                     % opakuj
read(X),                                   % načti další term
( X=end_of_file, !,                       % ukončení
  told, seen,                             % uzavření souborů
  see(In), tell(Out)                     % obnovení V/V
  ;                                       % není konec
  transformuj(X,Y)                       % vlastní zpracování
  write(Y),                             % term do F2
  fail                                   % návrat na začátek
) .                                       % cyklu
```

# Vstup a výstup: ISO

## Standard ISO

- `open(+Soubor, +Mode, ?Proud)`
  - » otevře `Soubor` v režimu `Mode` (`read`, `write`, `append`, `update`)
  - » proměnná `Proud` je vázána na číselnou identifikaci proudu
  - » atom `Proud` se stává identifikátorem proudu
  - » `open/4`
- `close(+Proud)`

# Vstup a výstup: ISO

## Standard ISO

- `set_input(+Stream)`

`open(file, read, Stream), set_input(Stream)`  
 $\cong$  `see(file)`

- `set_output(+Stream)`
- `current_input(-Stream)`
- `current_output(-Stream)`

# Zpracování přirozeného jazyka



## Eliza: Dialog s psychoanalytičkou

- J. Weizenbaum, ELIZA - A computer program for the study of natural language communication between man and machine.  
*Comm. of the ACM* **9** (1966), 36-45.
- Turingova imitační hra

```
eliza:- write('Hello. My name is Eliza. How can I help you?'),  
        nl, cti_vetu(V), eliza(V), !.
```

```
eliza(Vstup):-  
    member('quit', Vstup),  
    write('Goodbye. My secretary will send you a bill. '),  
    nl.
```

# Eliza: podnět a odezva

```
vzor([ 'I', am, 1 ],  
      [ 'How', long, have, you, been, 1, ? ] ).  
  
vzor([ 1, you, 2, me ],  
      [ 'What', makes, you, think, 'I', 2, you, ? ] ).  
  
vzor([ 'I', like, 1 ],  
      [ 'Does', anyone, else, in, your, family, like, 1, ? ] ).  
  
vzor([ 'I', feel, 1 ],  
      [ 'Do', you, often, feel, that, way, ? ] ).  
  
vzor([ 1, X, 2 ],  
      [ 'Can', you, tell, me, more, about, X, ? ] ) :- important(X).  
  
vzor([ 1 ], [ 'Please', go, 'on.' ] ).
```

# Eliza: pomocné predikáty

👉 Klíčová slova

```
important(father).
```

```
important(mother).
```

```
important(brother).
```

```
important(son).
```

```
important(daughter).
```

```
important(sister).
```

Predikát **hledej/3** pro hledání ve asociativním seznamu

```
hledej(Klic, [Klic-Hodnota|_], Hodnota).
```

```
hledej(Klic, [Klic1-_|Slovník], Hodnota):-  
    Klic \= Klic1,  
    hledej(Klic, Slovník, Hodnota).
```



# Eliza: komunikační smyčka

```
eliza(Vstup):-  
    vzor(Podnet,Reakce),  
    match(Podnet,Slovník,Vstup),  
    match(Reakce,Slovník,Vystup),  
    reply(Vystup),  
    cti_vetu(Vstup1),  
    !,  
    eliza(Vstup1).  
reply([H|T]):- write(H), write(' '), reply(T).  
reply([]):- nl.
```

# Eliza: reakce na podnět

```
match([Slovo|Vzor],Slovník,[Slovo|Cil]):-  
    atom(Slovo),  
    match(Vzor,Slovník,Cil).  
match([N|Vzor],Slovník,Cil):-  
    integer(N),  
    hledej(N,Slovník,LevyCil),  
    append(LevyCil,PravyCil,Cil),  
    match(Vzor,Slovník,PravyCil).  
match([],_,[]).
```

# Eliza: zpracování vstupu

```
% typ_znaku(+KodZnaku,?Typ):-  
% Typ znaku se zadany kodem KodZnaku,  
% Typ je oddelovac, konec_vety nebo jiny.  
typ_znaku(Z,konc_vety) :- member(Z,[33,46,63]), !.  
                                % vykřičník, tečka, otazník  
typ_znaku(Z,oddelovac)      :- Z =< 32, !.  
                                % mezery apod.  
typ_znaku(_,jiny).
```

# Eliza: načtení slova

```
% cti_slovo(+Pismeno,-S,-DalsiZnak):-  
%                               vrátí seznam S písmen slova,  
%                               které začíná Pismenem  
%                               a za ním následuje DalsiZnak.  
  
cti_slovo(Z,[ ],Z):-  
    typ_znaku(Z,konec_vety),!.    % konec věty  
  
cti_slovo(Z,[ ],Z):-  
    typ_znaku(Z,oddelovac),!.    % konec slova  
  
cti_slovo(Pis,[Pis|SezPis],DalsiZnak):-  
    get_code(Znak),  
    cti_slovo(Znak,SezPis,DalsiZnak).
```

# Eliza: načtení věty

```
% cti_vetu(-SeznamSlov):- přečte na vstupu větu  
%                               a vrátí SeznamSlov věty.
```

```
cti_vetu(SezSlov):-  
    get_code(Znak), cti_zbytek(Znak,SezSlov).
```

```
cti_zbytek(Z,[ ]):-  
    typ_znaku(Z,konec_vety), !.    % konec věty
```

```
cti_zbytek(Z,SezSlov):-  
    typ_znaku(Z,oddelovac), !,    % mezera apod.  
    cti_vetu(SezSlov).
```

```
cti_zbytek(Pismeno,[Slovo|SezSlov]):-  
    cti_slovo(Pismeno,SezPis,DalsiZnak),  
    name(Slovo,SezPis),  
    cti_zbytek(DalsiZnak,SezSlov).
```