Úvod do umělé inteligence, jazyk Prolog

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

➜ Organizace předmětu PB016

➜ Co je "umělá inteligence"

➜ Stručné shrnutí Prologu

Organizace předmětu PB016

Hodnocení předmětu:

➜ průběžná písemka (max 32 bodů)
- v první 1/2 semestru – 25. října v rámci přednášky
- pro každého jediný termín

➜ závěrečná písemka (max 96 bodů)
- dva řádné a jeden opravný termín

➜ hodnocení – součet bodů za obě písemky (max 128 bodů)

➜ známka A za více než 115 bodů známka E za více než 63 bodů

➜ rozdíly zk, k, z – různé limity

➜ *nově* – někteří můžou získat body za studentské referáty
- až 20 bodů – za kvalitní text (cca 5 stran) + 10–20 minut referát
- nutné *před průběžnou písemkou* domluvit téma – projekt/program, algoritmus z Náplně předmětu
- domluva *e-mailem* – návrh tématu, který musí projít schválením

Základní informace

➜ přednáška je nepovinná

➜ cvičení – samostudium, v rámci "třetího kreditu"

➜ web stránka předmětu – `http://nlp.fi.muni.cz/uui/`

➜ slajdy – průběžně doplňovány na webu předmětu

➜ kontakt na přednášejícího – Aleš Horák <`hales@fi.muni.cz`> (Subject: PB016 . . .)

➜ literatura:
- Russell, S. a Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd.ed., Prentice Hall, 2003. (prezenčně v knihovně)
- Bratko, I.: Prolog Programming for Artificial Intelligence, Addison-Wesley, 2001. (prezenčně v knihovně)
- slajdy na webu předmětu
- Jirků, Petr: Programování v jazyku Prolog, Praha : Státní nakladatelství technické literatury, 1991.

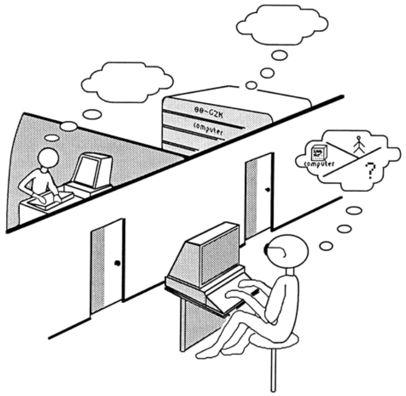
Náplň předmětu

1. jazyk Prolog **(20.9.)**
2. operace na datových strukturách **(27.9.)**
3. prohledávání stavového prostoru **(4.10.)**
4. heuristiky, best-first search, A* search **(11.10.)**
5. dekompozice problému, AND/OR grafy**(18.10.)**
6. problémy s omezujícími podmínkami, průběžná písemka **(25.10.)**
7. hry a základní herní strategie **(1.11.)**
8. inteligentní agenti, výroková logika, predikátová logika prvního řádu **(8.11.)**
9. TIL – transparentní intenzionální logika **(15.11.)**
10. reprezentace a vyvozování znalostí **(22.11.)**
11. studentské referáty **(29.11.)**
12. učení, rozhodovací stromy, neuronové sítě **(6.12.)**
13. zpracování přirozeného jazyka **(13.12.)**
14. předtermín **(20.12.)**

Co je “umělá inteligence”

systém, který se chová jako člověk Turingův test (1950)

zahrnuje:
- zpracování přirozeného jazyka (NLP)
- reprezentaci znalostí (KRepresentation)
- vyvozování znalostí (KReasoning)
- strojové učení
- (počítačové vidění)
- (robotiku)

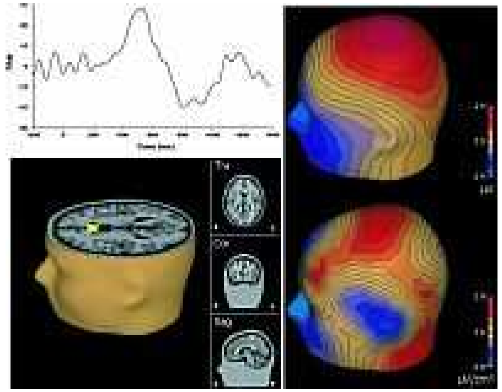


systém, který myslí jako člověk

- snaha porozumět postupům lidského myšlení – kognitivní (poznávací) věda
- využívá poznatků neurologie, neurochirurgie,. . .
  např.

  COLING 2000 – Angela Friederici:
  Language Processing in the Human Brain
  Max Planck Institute of Cognitive Neuroscience, Leipzig

  měření “Event Related Potentials” (ERP) v mozku – jako potvrzení oddělení syntaxe a sémantiky při zpracování věty



systém, který myslí rozumně od dob Aristotela (350 př.n.l.)

- náplň studia logiky
- problém – umět najít řešení teoreticky × prakticky (složitost a vyčíslitelnost)
- problém – neúplnost a nejistota vstupních dat

systém, který se chová rozumně inteligentní agent – systém, který

- jedná za nějakým účelem
- jedná samostatně
- na základě vstupů ze svého prostředí
- pracuje delší dobu
- adaptuje se na změny

Čím se budeme zabývat?

- základní struktury a algoritmy běžně používané při technikách programovaní pro inteligentní agenty
- strategie řešení, prohledávání stavového prostoru, heuristiky, . . .
- s příklady v jazyce Prolog

Stručné shrnutí Prologu

Historie:

- 70. l. Colmerauer, Kowalski; D.H.D. Warren (WAM); ⟶ CLP, paralelní systémy
- PROgramování v LOGice; část predikátové logiky prvního řádu (logika Hornových klauzulí)
- deklarativnost (specifikace programu je přímo programem)
- řešení problémů týkajících se objektů a vztahů mezi nimi

Prology na FI:

- SICStus Prolog (modul sicstus)
- SWI (modul pl)
- ECLiPSe (modul eclipse)
- stroje aisa, erinys, oreias, nymfe
- verze

Principy

- backtracking řízený unifikací, hojně využívá rekurzi
- spojitost s logikou: snaha dokázat pravdivost daného cíle; cíl je dokázán, unifikuje-li s hlavou nějaké klauzule a všechny podcíle v těle této klauzule jsou rovněž dokázány. Strategie výběru podcíle: shora dolů, zleva doprava.
- unifikace: řídicí mechanismus, hledání nejobecnějšího unifikátoru dvou termů. Např.

  informace(*Manzel*,dana,*Deti*,svatba('20.12.1940')) = informace(petr,dana,[jan,pavel],*Info*).

  po unifikaci: ***Manzel*=petr**, ***Deti*=[jan,pavel]**, ***Info*=svatba('20.12.1940')**
- backtracking: standardní metoda prohledávání stavového prostoru do hloubky (průchod stromem ⟶ nesplnitelný cíl ⟶ návrat k nejbližšímu minulému bodu s alternativní volbou)
- rekurze

```
potomek(X,Y):– rodic(Y,X).
potomek(X,Y):– rodic(Z,X), potomek(Z,Y).
```


Syntax jazyka Prolog

logický (prologovský) program – seznam klauzulí (pravidel a faktů) – nikoli *množina*

klauzule – seznam literálů

- Literál před **:-** je hlava, ostatní literály tvoří tělo klauzule.
- Význam klauzule je implikace:
  - **hlava:-tělo1, tělo2, ...**
  - **tělo1** ∧ **tělo2** ∧ . . . ⇒ **hlava**
  - *Pokud je splněno* **tělo1** *a současně* **tělo2** *a současně . . . , pak platí také* **hlava**.
- 3 možné typy klauzulí:
  - fakt: hlava bez těla. Zápis v Prologu: **p(X,Y).** (ekv. *p(X,Y):-true.*)
  - pravidlo: hlava i tělo. Prolog: **p(Z,X) :- p(X,Y), p(Y,Z).**
  - cíl: tělo bez hlavy. Prolog: **?- p(g,f).**

predikát – seznam (všech) klauzulí se stejným funktorem a aritou v hlavovém literálu.

- Zapisuje se ve tvaru *funktor*/*arita* – **potomek/2**.

literál – atomická formule, nebo její negace

atomická formule – v Prologu zcela odpovídá složenému termu (syntaktický rozdíl neexistuje)

term:

- konstanta: **a**, **1**, **'.'**, **[]**, **sc2**

  atomic/1 (metalogické testování na konstantu)

  atom/1, **number/1**
- proměnná: **X, Vys, _**

  var/1 (metalogické testování na proměnnou)
- složený term: **f(a,X)**

  funktor, argumenty, arita

  functor/3 dává funktor termu, **arg/3** dává n-tý argument

  zkratka pro zápis seznamů:

  [1,a,b3] odpovídá struktuře **'.'(1, '.'(a, '.'(b3, [])))**

Příklad

jednoduchý příklad – DB rodinných vztahů:

```
otec(milan,dana).
otec(milan,petr).
otec(jan,david).

matka(pavla,dana).
matka(pavla,petr).
matka(jana,david).

rodic(X,Y):- otec(X,Y).
rodic(X,Y):- matka(X,Y).

?- otec(X,dana).        ?- rodic(X,david).
X = milan               X = jan ;
Yes                     X = jana ;
```

fakty (DB) – otec(…), matka(…)

pravidla – rodic(…)

Příklad

predikát **sourozenci(X,Y)** – je **true**, když **X** a **Y** jsou sourozenci.

```
sourozenci(X,Y):- otec(O,X), otec(O,Y), X\=Y, matka(M,X), matka(M,Y).
```

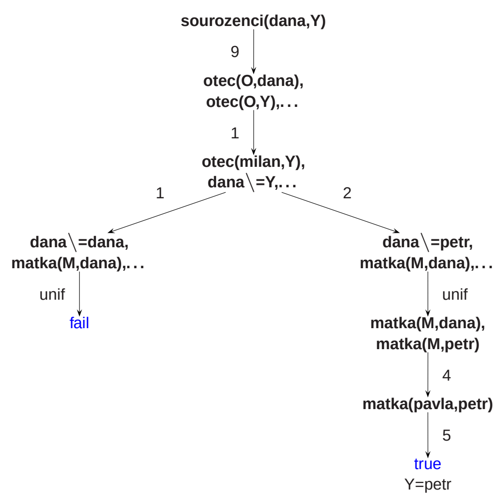
```
1 otec(milan,dana).
2 otec(milan,petr).
3 otec(jan,david).
4 matka(pavla,dana).
5 matka(pavla,petr).
6 matka(jana,david).
7 rodic(X,Y):- otec(X,Y).
8 rodic(X,Y):- matka(X,Y).
```

```
?- sourozenci(dana,Y).
1, otec(O,dana)         % O = milan
2, otec(milan,Y)        % Y = dana
3, dana \= dana         % fail -> backtracking
2*, otec(milan,Y)       % Y = petr
3, matka(M,dana)        % M = pavla
4, matka(pavla,petr)    % true

Y = petr

Yes
```


Strom výpočtu

Dotaz **?- sourozenci(dana,Y)**.

```
1  otec(milan,dana).
2  otec(milan,petr).
3  otec(jan,david).
4  matka(pavla,dana).
5  matka(pavla,petr).
6  matka(jana,david).
7  rodic(X,Y):- otec(X,Y).
8  rodic(X,Y):- matka(X,Y).
9  sourozenci(X,Y):- otec(O,X), otec(O,Y), X\=Y,
                     matka(M,X), matka(M,Y).
```



Rozdíly od procedurálních jazyků

- ➜ single assignment
- ➜ **=** (unifikace) vs. přiřazovací příkaz, **==** (identita), **is** (vyhodnocení aritm. výrazu). rozdíly:

```
?- A=1, A=B.   % B=1 Yes
?- A=1, A==B.  % No
?- A=1, B is A+1.  % B=2 Yes
```

- ➜ vícesměrnost predikátů (omezená, obzvláště při použití řezu)

```
?- otec(X,dana).
?- otec(milan,X).
?- otec(X,Y).
```

(rozlišení vstupních/výstupních proměnných: **+ - ?**)

- ➜ cykly, podmíněné příkazy

```
tiskniseznam(S)      :- write('seznam=['),nl,tiskniseznam(S,1).
tiskniseznam([],_)   :- write(']'),nl.
tiskniseznam([H|T],N):- tab(4),write(N),write(': '),write(H),nl,N1 is N+1,tiskniseznam(T,N1).
```

Programujeme

```
consult('program.pl').          % " kompiluj " program.pl
['program.pl',program2].        % " kompiluj " program.pl, program2.pl
listing.                        % vypiš programové predikáty
trace, rodic(X,david).          % trasuj volání predikátu
notrace.                        % zruš režim trasování
halt.                           % ukonči interpret
```


Fibonacciho čísla

Fibonacciho čísla jsou čísla z řady: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Rekurenční vzorec této řady je:

$$\mathrm{fib}_0 = 0$$
$$\mathrm{fib}_1 = 1$$
$$\mathrm{fib}_i = \mathrm{fib}_{i-1} + \mathrm{fib}_{i-2}, \text{pro } i \geq 2$$

Přepis do Prologu je přímočarý:

```
fib(0,0).
fib(1,1).
fib(X,Y) :- X1 is X-1, X2 is X-2, fib(X1,Y1), fib(X2,Y2), Y is Y1+Y2.
```


Fibonacciho čísla II

Předchozí program – exponenciální časová složitost (konstatní paměťová)

Využití extralogických predikátů – lineární časová složitost (a lineární paměťová)

```
fib(0,0).
fib(1,1).
fib(X,Y) :- X1 is X-1, X2 is X-2, fib(X1,Y1), fib(X2,Y2), Y is Y1+Y2, asserta(fib(X,Y)).
```

Operace na datových strukturách

Aleš Horák

E-mail: hales@fi.muni.cz

http://nlp.fi.muni.cz/uui/

Obsah:

➜ Práce se seznamy
➜ Binární stromy
➜ Reprezentace grafů

Operace na datových strukturách

Seznam:

➜ rekurzivní datová struktura

➜ uspořádaná posloupnost prvků (libovolných termů včetně seznamů)

➜ operátor **./2**; prázdný seznam **[]**

➜ **.(Hlava,Tělo)**, alternativně **[Hlava|Tělo]**, **Hlava** je (typu) *prvek seznamu*, **Tělo** je (typu) *seznam*

.(a,[])	**[a]**	**[a|[]]**
.(a,.(b,.(c,[])))	**[a,b,c]**	**[a,b|[c]], [a|[b,c]], [a,b,c|[]], [a|[b,c|[]]], [a|[b|[c|[]]]]**
...	**[a1,[[b3,c3],d2,e2],f1]**	...

Práce se seznamy – member

member(+Prvek,+Seznam) – **true**, pokud v seznamu existuje zadaný prvek

1.

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
?- member(a,[X,b,c]).
     X=a
     Yes
```

2.

```
member(X,[Y|_]) :- X == Y.
member(X,[_|T]) :- member(X,T).
?- member(a,[X,b,c]).          ?- member(a,[a,b,a]),write(ok),nl,fail .
     No                             ok
                                    ok
                                    No
```

3.

```
member(X,[Y|_]) :- X == Y.
member(X,[Y|T]) :- X \== Y, member(X,T).
?- member(a,[a,b,a]),write(ok),nl,fail .
     ok
     No
```


Práce se seznamy – del a insert

predikát **del(A,L,Vysl)** smaže všechny výskyty prvku **A** ze seznamu **L**

del1(A,L,Vysl) smaže vždy jeden (podle pořadí) výskyt prvku **A** v seznamu **L**

```
del(_,[],[]).                                  ?- del (1,[1,2,1,1,2,3,1,1], L).
del(A,[A|T],V) :- del(A,T,V).                       L = [2, 2, 3]
del(A,[H|T1],[H|T2]) :- A\=H, del(A,T1,T2).         Yes
                                               ?- del1 (1,[1,2,1],L).
del1(A,[A|T],T).                                    L = [2, 1] ;
del1(A,[H|T1],[H|T2]) :- del1(A,T1,T2).             L = [1, 2] ;
                                                    No
```

insert(A,L,Vysl) vkládá postupně (při žádosti o další řešení) na všechny pozice seznamu **L** prvek **A**

jednoduchý **insert1(A,L,Vysl)** vloží **A** na začátek seznamu **L** (ve výsledku **Vysl**)

```
insert (A,L,[A|L]).                            ?- insert (4,[2,3,1], L).
insert (A,[H|T1],[H|T2]):- insert(A,T1,T2).         L = [4, 2, 3, 1] ;
                                                    L = [2, 4, 3, 1] ;
                                                    L = [2, 3, 4, 1] ;
insert1 (X,List ,[X|List ]).                        L = [2, 3, 1, 4] ;
                                                    No
```

PRÁCE SE SEZNAMY – PERMUTACE

1. pomocí **insert**

```
perm1 ([],[]).                                   ?– perm1([1,2,3],L).
perm1([H|T],L):– perm1(T,V), insert(H,V,L).          L = [1, 2, 3] ;
                                                     L = [2, 1, 3] ;
                                                     L = [2, 3, 1] ;
                                                     L = [1, 3, 2] ;
                                                     L = [3, 1, 2] ;
                                                     L = [3, 2, 1] ;
                                                     No
```

2. pomocí **del1**

```
perm2 ([],[]).
perm2(L,[X|P]) :– del1(X,L,L1),perm2(L1,P).
```

3. pomocí **append**

```
perm3 ([],[]).
perm3(L,[H|T]):– append(A,[H|B],L),append(A,B,L1), perm3(L1,T).
```


PRÁCE SE SEZNAMY – append

append(?Seznam1,?Seznam2,?Seznam) – **Seznam** je spojení seznamů **Seznam1** a **Seznam2**

```
append([],L,L).
append([H|T1],L2,[H|T]) :– append(T1,L2,T).
```

predikát **append** je vícesměrný:

```
?– append([a,b],[c,d],L).
   L = [a, b, c, d]
   Yes
?– append(X,[c,d],[a,b,c,d ]).
   X = [a, b]
   Yes
?– append(X,Y,[a,b,c]).
   X = []          Y = [a, b, c];
   X = [a]         Y = [b, c];
   X = [a, b]      Y = [c];
   X = [a, b, c]   Y = [];
   No
```


PRÁCE SE SEZNAMY – VYUŽITÍ append

predikát **append** je všestranně použitelný:

```
member(X,Ys)         :– append(As,[X|Xs],Ys).
 last (X,Xs)         :– append(As,[X],Xs).
 prefix (Xs,Ys)      :– append(Xs,As,Ys).
 suffix (Xs,Ys)      :– append(As,Xs,Ys).
 sublist (Xs,AsXsBs) :– append(AsXs,Bs,AsXsBs), append(As,Xs,AsXs).
adjacent(X,Y,Zs)     :– append(As,[X,Y|Ys],Zs).
```


PRÁCE SE SEZNAMY – EFEKTIVITA append

Efektivní řešení predikátu **append** – rozdílové seznamy (difference lists)

Rozdílový seznam se zapisuje jako **Seznam1-Seznam2**.

Např.: **[a,b,c]** . . . **[a,b,c] - []** nebo **[a,b,c,d] - [d]** nebo **[a,b,c,d,e] - [d,e]**, obecně **[a,b,c|X] - X**
[] . . . **A-A**
[a] . . . **[a|A]-A**

Seznam2 jako volná proměnná slouží jako "ukazatel" na konec seznamu **Seznam1**

predikát **append** s rozdílovými seznamy (**append_dl**):

```
append_dl(A–B,B–C,A–C).

?– append_dl([a,b|X]–X,[c,d|Y]–Y,Z).
X = [c, d|Y]
Y = Y
Z = [a, b, c, d|Y] – Y
Yes
```

Třídění seznamů — quicksort

predikát **qsort(L,Vysl)** – třídí seznam **L** technikou rozděl a panuj

L=[5,3,7,8,1,4,7,6] — **L=[H|T],H=5**

↓

T=[3,7,8,1,4,7,6] — **divide(5, . . .)**

↙ ↘

$\forall$prvky ≤ 5 **M=[3,1,4], qsort(M)** | $\forall$prvky > 5 **V=[7,8,7,6], qsort(V)**

↓ | ↓

M1=[1,3,4] | **V1=[6,7,7,8]** — **append - M1.[5].V1**

↘ ↙

Vysl=[1,3,4,5,6,7,7,8]

Třídění seznamů — quicksort

predikát **qsort(L,Vysl)** – třídí seznam **L** technikou rozděl a panuj

```
qsort ([],[])    :- !.
qsort([H],[H])  :- !.
qsort([H|T],L)  :- divide(H,T,M,V),
                   qsort(M,M1), qsort(V,V1),
                   append(M1,[H|V1],L).

divide(_ ,[],[],[])    :- !.
divide(H,[K|T],[K|M],V) :- K=<H, !, divide(H,T,M,V).
divide(H,[K|T],M,[K|V]) :- K>H, divide(H,T,M,V).
```


Třídění seznamů — quicksort II

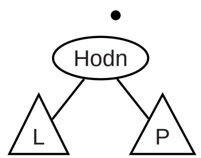
predikát **qsort_dl(L,Vysl)** – efektivnější varianta predikátu **qsort** s rozdílovými seznamy

```
qsort(L,S):- qsort_dl(L,S-[]).

qsort_dl ([], A-A).
qsort_dl ([H|T],A-B):- divide(H,T,L1,L2),
                        qsort_dl (L2,A1-B),
                        qsort_dl (L1,A-[H|A1]).

divide(_ ,[],[],[])   :- !.
divide(H,[K|T],[K|M],V):- K=<H, !, divide(H,T,M,V).
divide(H,[K|T],M,[K|V]):- K>H, divide(H,T,M,V).
```


Uspořádané binární stromy

Reprezentace binárního stromu:

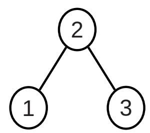
- ➜ **nil** – prázdný strom
- ➜ **t(L,Hodn,P)** – strom



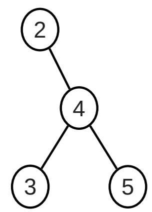
Příklady stromů:

t(nil,8,nil)

t(t(nil,1,nil),2,t(nil,3,nil))



t(nil,2,t(t(nil,3,nil),4,t(nil,5,nil)))



Přidávání do binárního stromu

addleaf(T,X,Vysl) přidá do binárního stromu **T** hodnotu **X** na správnou pozici vzhledem k setřídění 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).

?- addleaf(nil,6,T),addleaf(T,8,T1), addleaf(T1,2,T2), addleaf(T2,4,T3), addleaf(T3,1,T4).
?- addleaf(t(t(t( nil ,1, nil ),2, t(t( nil ,3, nil ),4, t( nil ,5, nil ))),
      6,t(t( nil ,7, nil ),8, t( nil ,9, nil ))), 10, T).
```

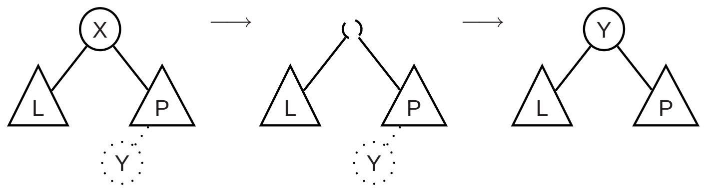
Predikát **addleaf** není vícesměrný ☹ ⇒ nelze definovat:

```
del(T,X,T1) :- addleaf(T1,X,T).
```


Odebírání z binárního stromu

- ➜ pokud je odebíraná hodnota v listu ⟶ nahradí se hodnotou **nil**
- ➜ jestliže je ale v kořenu (pod)stromu ⟶ je nutné tento (pod)strom přestavět

Přestavba binárního stromu při odstraňování kořene **X**:



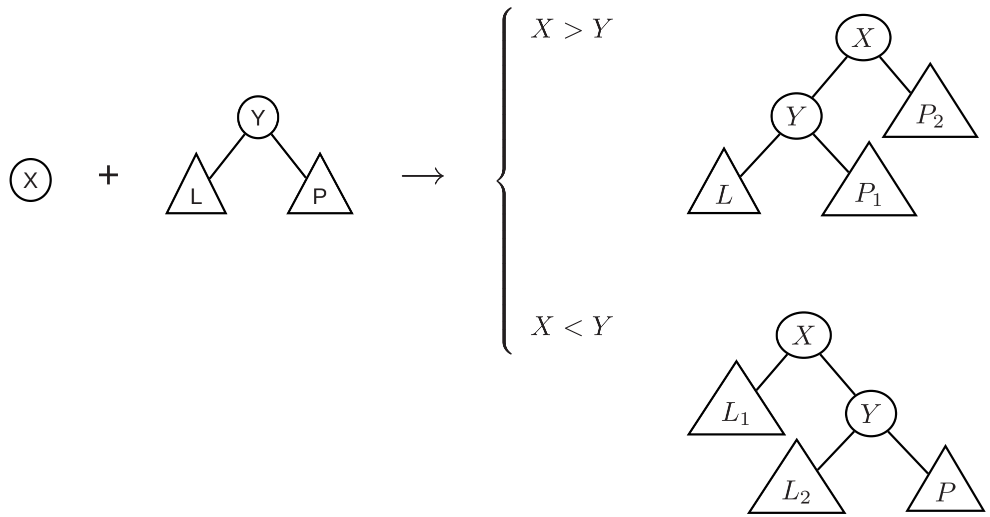
Odebírání z binárního stromu

delleaf(T,X,Vysl) odstraní ze stromu **T** uzel s hodnotou **X**

```
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).
```


Vícesměrný algoritmus pro vkládání/odebírání

Jiný způsob vkládání:



Vícesměrný algoritmus pro vkládání/odebírání

add(T,X,Vysl) přidá do binárního stromu **T** uzel s hodnotou **X** jako kořen s přeuspořádáním stromu

```
add(T,X,T1) :- addroot(T,X,T1).
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)).
```

Definice predikátu **gt(X,Y)** – na konečném uživateli.

Funguje i "obráceně" ⇒ lze definovat:

```
del(T,X,T1) :- add(T1,X,T).
```

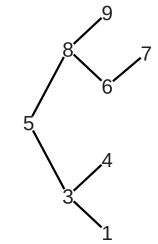

Výpis binárního stromu

pomocí odsazení zobrazujeme úroveň uzlu ve stromu a celkové uspořádání uzlů (strom je tedy zobrazen "naležato")

```
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)))
```

⟶

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



show(T) vypíše obsah uzlů stromu **T** se správným odsazením

```
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).
```

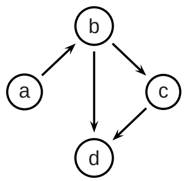

Reprezentace grafů

Příklady způsobů reprezentace grafů (v Prologu):

1. 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.

```
graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)]).
```

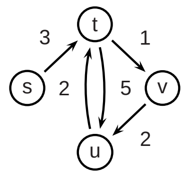


znázorňuje orientovaný graf

2. **digraph(V,E)** definuje uspořádanou dvojici seznamů vrcholů (**V**) a hran (**E**).

   Hrany jsou tvaru **a(PocatecniV,KoncovyV,CenaHrany)**.

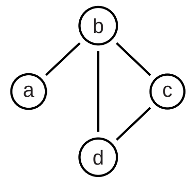
```
digraph([s,t,u,v],[a(s,t,3),a(t,v,1),a(t,u,5),a(u,t,2),a(v,u,2)]).
```



znázorňuje orientovaný ohodnocený graf

3. graf může být uložen v programové databázi jako posloupnost faktů (i pravidel).

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



díky přidanému pravidlu představuje neorientovaný graf (bez pravidla je orientovaný).

Cesty v grafech

Cesta v neorientovaném grafu:

path(A,Z,G,C) v grafu **G** najde z vrcholu **A** do vrcholu **Z** cestu **C** (**G** je ve tvaru 1).

```
path(A,Z,Graph,Cesta) :− path1(A,[Z],Graph,Cesta).

path1(A,[A|Cesta1],_,[A|Cesta1]).
path1(A,[Y|Cesta1],Graph,Cesta) :− adjacent(X,Y,Graph),not(member(X,Cesta1)),
      path1(A,[X,Y|Cesta1],Graph,Cesta).

adjacent(X,Y,graph(Nodes,Edges)) :− member(e(X,Y),Edges);member(e(Y,X),Edges).
```


Cesty v grafech II

Cesta v ohodnoceném neorientovaném grafu:

path(A,Z,Graf,Cesta,Cena) hledá libovolnou cestu z jednoho vrcholu do druhého a její cenu v ohodnoceném neorientovaném grafu.

```
path(A,Z,Graf,Cesta,Cena) :− path1(A,[Z],0,Graf,Cesta,Cena).

path1(A,[A|Cesta1],Cena1,Graf,[A|Cesta1],Cena1).
path1(A,[Y|Cesta1],Cena1,Graf,Cesta,Cena) :− adjacent(X,Y,CenaXY,Graf),
      not(member(X,Cesta1)),Cena2 is Cena1+CenaXY,
      path1(A,[X,Y|Cesta1],Cena2,Graf,Cesta,Cena).

adjacent(X,Y,CenaXY,Graf) :− member(X−Y/CenaXY,Graf);member(Y−X/CenaXY,Graf).
```

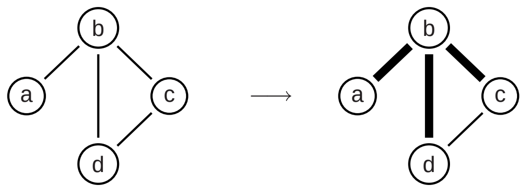
Graph je seznam hran ve tvaru **X-Y/CenaXY** (viz **adjacent**).

Kostra grafu

Kostra grafu je strom, který prochází všechny vrcholy grafu a jehož hrany jsou zároveň hranami grafu.

```
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).
```

```
?− stree([a−b,b−c,b−d,c−d],T).
   S = [b−d, b−c, a−b]
   Yes
```



Prohledávání stavového prostoru

Aleš Horák

E-mail: **hales@fi.muni.cz**

http://nlp.fi.muni.cz/uui/

Obsah:

➜ Problém osmi dam

➜ Prohledávání stavového prostoru

➜ Prohledávání do hloubky

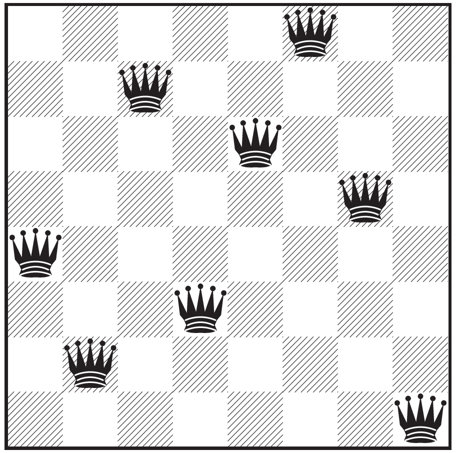
➜ Prohledávání do šířky

➜ Prohledávání s postupným prohlubováním

➜ Shrnutí vlastností algoritmů neinformovaného prohledávání

Problém osmi dam

úkol: *Rozestavte po šachovnici 8 dam tak, aby se žádné dvě vzájemně neohrožovaly.*



celkem pro *8 dam* existuje *92 různých řešení*

Problém osmi dam I

datová struktura – osmiprvkový seznam **[X1/Y1, X2/Y2, X3/Y3, X4/Y4, X5/Y5, X6/Y6, X7/Y7, X8/Y8]**

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

```
solution(S) :- template(S), sol(S).

sol([]).
sol([X/Y|Others]) :- sol(Others),
                     member(X,[1,2,3,4,5,6,7,8]),
                     member(Y,[1,2,3,4,5,6,7,8]),
                     noattack(X/Y,Others).

noattack(_,[]).
noattack(X/Y,[X1/Y1|Others]) :- X=\=X1, Y=\=Y1, Y1-Y=\=X1-X, Y1-Y=\=X-X1,
                                noattack(X/Y,Others).

template([X1/Y1, X2/Y2, X3/Y3, X4/Y4, X5/Y5, X6/Y6, X7/Y7, X8/Y8]).

?- solution(Solution).
   Solution = [8/4, 7/2, 6/7, 5/3, 4/6, 3/8, 2/5, 1/1] ;
   Solution = [7/2, 8/4, 6/7, 5/3, 4/6, 3/8, 2/5, 1/1] ;
   Yes
```


Problém osmi dam II

počet možností u řešení I $= 64 \cdot 63 \cdot 62 \ldots \cdot 57 \approx 1.8 \times 10^{14}$

omezení stavového prostoru – každá dáma má svůj sloupec

počet možností u řešení II $= 8 \cdot 7 \cdot 6 \ldots \cdot 1 = 40\,320$

```
solution(S) :- template(S), sol(S).

sol([]).
sol([X/Y|Others]) :- sol(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]).
```

Problém osmi dam III

k souřadnicím x a y $\longrightarrow$ přidáme i souřadnice diagonály u a v

$$u = x - v \qquad D_x = [1..8] \longrightarrow D_u = [-7..7]$$
$$v = x + v \qquad D_y = [1..8] \qquad D_v = [2..16]$$

po každém umístění dámy aktualizujeme seznamy volných pozic počet možností u řešení III $= 2\,057$

```
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).

del(Item,[Item|List], List).
del(Item,[First|List],[First|List1]) :- del(Item,List,List1).
```

Problém n dam pro $n = 100$: řešení I $\ldots 10^{400}$ řešení II $\ldots 10^{158}$ řešení III $\ldots 10^{52}$

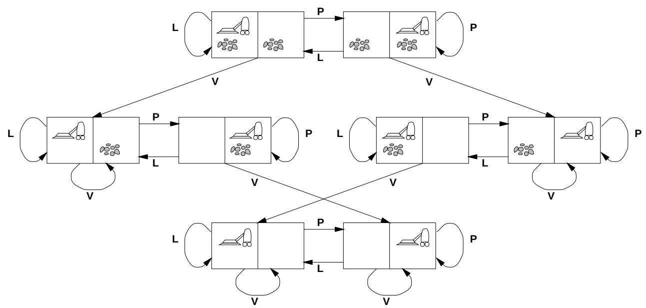
Prohledávání stavového prostoru

Řešení problému prohledáváním stavového prostoru:

- předpoklady – statické a deterministické prostředí, diskrétní stavy
- *stavový prostor*
- *počáteční stav* **init(State)**
- *cílová podmínka* **goal(State)**
- *přechodové akce* **move(State,NewState)**
- *prohledávací strategie*

Problém agenta Vysavače:

- máme dvě místnosti (L, P)
- jeden vysavač (v L nebo P)
- v každé místnosti je/není špína
- počet stavů je $2 \times 2^2 = 8$
- akce $=\{doLeva, doPrava, Vysávej\}$



Abstrakce prohledávání stavového prostoru

- *prohledávací strom*
- *kořenový uzel*
- *uzel* prohledávacího stromu:
  - *stav*
  - *rodičovský uzel*
  - *přechodová akce*
  - *hloubka uzlu*
  - *cena* – $g(n)$ cesty, $c(x, a, y)$ přechodu
- *(optimální) řešení*

Další příklad – posunovačka

počáteční stav (např.)

7	2	4
5		6
8	3	1

$\longrightarrow \ldots \longrightarrow$

cílový stav

	1	2
3	4	5
6	7	8

- hra na čtvercové šachovnici $m \times m$ s $n = m^2 - 1$ očíslovanými kameny
- příklad pro šachovnici 3×3, posunování osmi kamenů (8-posunovačka)
- stavy – pozice včech kamenů
- akce – "pohyb" prázdného místa

☞ Optimální řešení obecné n-posunovačky je NP-úplné

Počet stavů	u 8-posunovačky	$\ldots\ 9!/2 = 181\,440$
	u 15-posunovačky	$\ldots\ 10^{13}$
	u 24-posunovačky	$\ldots\ 10^{25}$

Reálné problémy řešitelné prohledáváním

- hledání cesty z města A do města B
- hledání itineráře
- problém obchodního cestujícího
- návrh VLSI čipu
- navigace auta, robota, ...
- postup práce automatické výrobní linky
- návrh proteinů – 3D-sekvence aminokyselin
- Internetové vyhledávání informací

Řešení problému prohledáváním

Kostra algoritmu:

```
solution(Solution) :- init(State),solve(State,Solution).

solve(State,[State]) :- goal(State).
solve(State,[State|Sol]) :- move(State,NewState),solve(NewState,Sol).
```

move(State,NewState) – definuje prohledávací strategii

Porovnání strategií:

- úplnost
- optimálnost
- časová složitost
- prostorová složitost

složitost závisí na:

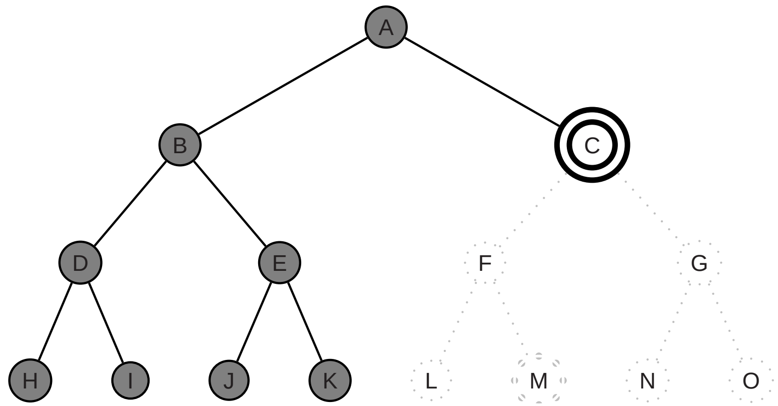
- b – faktor větvení (branching factor)
- d – hloubka cíle (goal depth)
- m – maximální hloubka větve/délka cesty (maximum depth/path)

Neinformované prohledávání

- prohledávání do hloubky
- prohledávání do hloubky s limitem
- prohledávání do šířky
- prohledávání podle ceny
- prohledávání s postupným prohlubováním

Prohledávání do hloubky

Prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (*Depth-first Search, DFS*)



Prohledávání do hloubky

procedurální programovací jazyk – uzly se uloží do zásobníku (fronty LIFO) × Prolog – využití rekurze

```
solution(Node,Solution) :- depth_first_search([], Node,Solution).

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


Prohledávání do hloubky – vlastnosti

úplnost	není úplný (nekonečná větev, cykly)
optimálnost	není optimální
časová složitost	$O(b^m)$
prostorová složitost	$O(bm)$, lineární

Největší problém – nekonečná větev = nenajde se cíl, program neskončí!

Prohledávání do hloubky s limitem

Řešení nekonečné větve – použití "zarážky" = limit hloubky ℓ

```
solution(Node,Solution) :- depth_first_search_limit(Node,Solution,ℓ).

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

neúspěch (**fail**) má dvě možné interpretace – vyčerpání limitu nebo neexistenci řešení

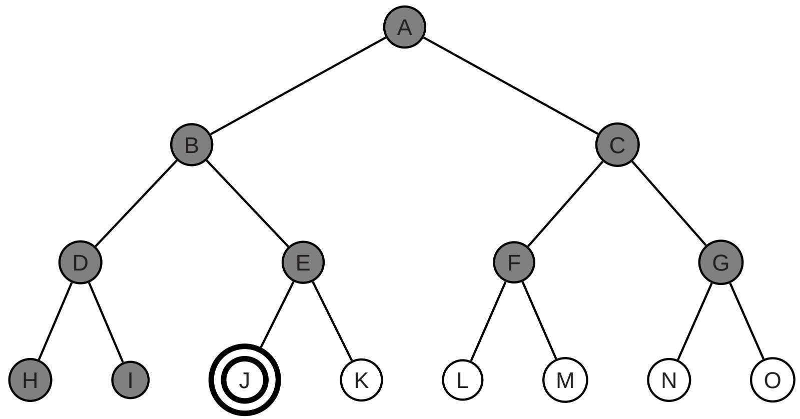
Vlastnosti:

úplnost	není úplný (pro $\ell < d$)
optimálnost	není optimální (pro $\ell > d$)
časová složitost	$O(b^\ell)$
prostorová složitost	$O(b\ell)$

dobrá volba limitu ℓ – podle znalosti problému

Prohledávání do šířky

Prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou. (*Breadth-first Search, BFS*)



PROHLEDÁVÁNÍ DO ŠÍŘKY

procedurální programovací jazyk – uzly se uloží do fronty (FIFO) × Prolog – udržuje seznam cest

```
solution(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], (move(N,M),not(member(M,[N|Path]))), NewPaths),
    NewPaths\=[], append(Paths,NewPaths,Path1), !,
    breadth_first_search(Path1,Solution); breadth_first_search(Paths,Solution).
```

bagof(+Prom,+Cíl,-Sezn) *postupně vyhodnocuje* ***Cíl*** *a všechny vyhovující instance* ***Prom*** *řadí do seznamu* ***Sezn***

p :- a,b;c. ⇔ p :- (a,b);c.

Vylepšení:

- **append** ⟶ **append_dl**
- seznam cest:

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

PROHLEDÁVÁNÍ DO ŠÍŘKY – VLASTNOSTI

úplnost	je úplný (pro konečné b)
optimálnost	je optimální podle délky cesty/není optimální podle obecné ceny
časová složitost	$1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, exponenciální v d
prostorová složitost	$O(b^{d+1})$ (každý uzel v paměti)

Největší problém – paměť:

Hloubka	Uzlů	Čas	Paměť
2	1100	0.11 sek	1 MB
4	111 100	11 sek	106 MB
6	10^7	19 min	10 GB
8	10^9	31 hod	1 TB
10	10^{11}	129 dnů	101 TB
12	10^{13}	35 let	10 PB
14	10^{15}	3 523 let	1 EB

Ani čas není dobrý ⟶ potřebujeme informované strategie prohledávání.

PROHLEDÁVÁNÍ PODLE CENY

- BFS je optimální pro rovnoměrně ohodnocené stromy × prohledávání podle ceny (Uniform-cost Search) je optimální pro obecné ohodnocení
- fronta uzlů se udržuje uspořádaná podle ceny cesty

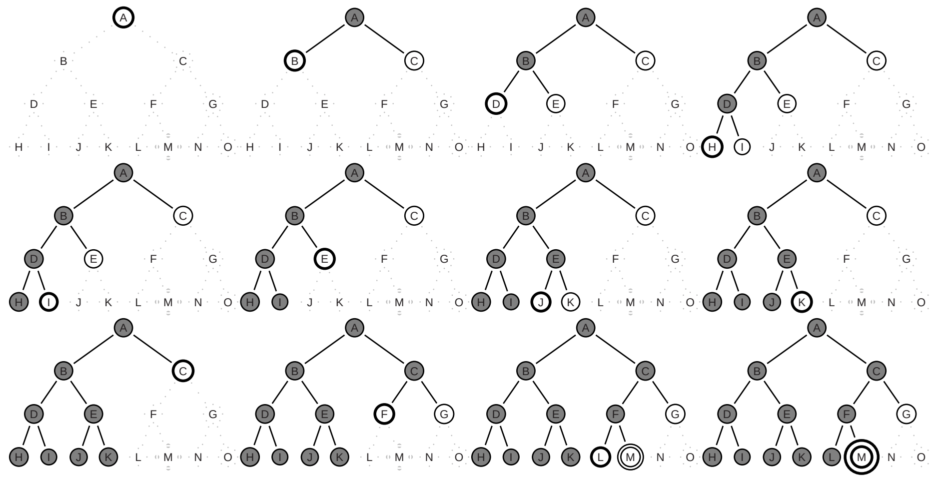
Vlastnosti:

úplnost	je úplný (pro cena $\geq \epsilon$)
optimálnost	je optimální (pro cena $\geq \epsilon$, $g(n)$ roste)
časová složitost	počet uzlů s $g \leq C^*$, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, kde $C^* \ldots$ cena optimálního řešení
prostorová složitost	počet uzlů s $g \leq C^*$, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

PROHLEDÁVÁNÍ S POSTUPNÝM PROHLUBOVÁNÍM

prohledávání do hloubky s postupně se zvyšujícím limitem (Iterative deepening DFS, IDS)

limit=3



Prohledávání s postupným prohlubováním – vlastnosti

úplnost	je úplný (pro konečné b)
optimálnost	je optimální (pro $g(n)$ rovnoměrně neklesající funkce hloubky)
časová složitost	$d(b) + (d-1)b^2 + \ldots + 1(b^d) = O(b^d)$
prostorová složitost	$O(bd)$

➜ kombinuje výhody BFS a DFS:
- nízké paměťové nároky – lineární
- optimálnost, úplnost

➜ zdánlivé plýtvání opakovaným generováním
ALE generuje o jednu úroveň míň, např. pro $b = 10, d = 5$:

$$\begin{array}{rcll} N(\mathrm{IDS}) &=& 50 + 400 + 3\,000 + 20\,000 + 100\,000 &= 123\,450 \\ N(\mathrm{BFS}) &=& 10 + 100 + 1\,000 + 10\,000 + 100\,000 + 999\,990 &= 1\,111\,100 \end{array}$$

IDS je nejvhodnější neinformovaná strategie pro velké prostory a neznámou hloubku řešení.

Shrnutí vlastností algoritmů neinformovaného prohledávání

Vlastnost	*do hloubky*	*do hloubky s limitem*	*do šířky*	*podle ceny*	*s postupným prohlubováním*
úplnost	ne	ano, pro $l \geq d$	ano*	ano*	ano*
optimálnost	ne	ne	ano*	ano*	ano*
časová složitost	$O(b^m)$	$O(b^\ell)$	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^d)$
prostorová složitost	$O(bm)$	$O(b\ell)$	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bd)$

Heuristiky, best-first search, A* search

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

➜ Informované prohledávání stavového prostoru

➜ Heuristické hledání nejlepší cesty

➜ Příklad – řešení posunovačky

➜ Příklad – rozvrh práce procesorů

INFORMOVANÉ PROHLEDÁVÁNÍ STAVOVÉHO PROSTORU

Neinformované prohledávání:

➜ DFS, BFS a varianty

➜ nemá (téměř) žádné informace o pozici cíle – slepé prohledávání

➜ zná pouze:

- počáteční/cílový stav
- přechodovou funkci

Informované prohledávání:

má navíc informaci o (odhadu) blízkosti stavu k cílovému stavu – heuristická funkce (heuristika)

HEURISTICKÉ HLEDÁNÍ NEJLEPŠÍ CESTY

➜ Best-first Search

➜ použití ohodnocovací funkce $f(n)$ pro každý uzel – výpočet přínosu daného uzlu

➜ udržujeme seznam uzlů uspořádaný (vzestupně) vzhledem k $f(n)$

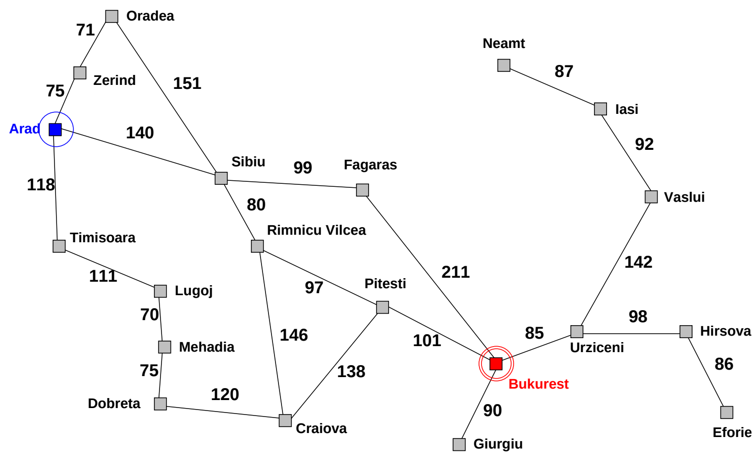
➜ použití heuristické funkce $h(n)$ pro každý uzel – odhad vzdálenosti daného uzlu od cíle

➜ čím *menší* $h(n)$, tím blíže k cíli, $h(\mathrm{Goal}) = 0$.

➜ nejjednodušší varianta – hladové heuristické hledání, *Greedy best-first search*

$f(n) = h(n)$

SCHÉMA RUMUNSKÝCH MĚST

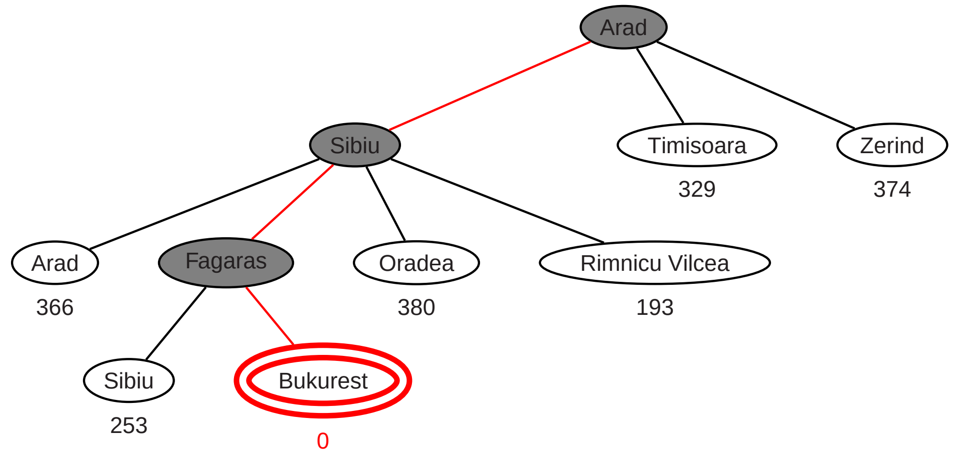


Arad	366
Bukurest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vilcea	199
Zerind	374

HLADOVÉ HEURISTICKÉ HLEDÁNÍ – PŘÍKLAD

Hledání cesty z města *Arad* do města *Bukurest*

ohodnocovací funkce $f(n) = h(n) = h_{\text{vzd\_Buk}}(n)$, přímá vzdálenost z n do Bukuresti



HLADOVÉ HEURISTICKÉ HLEDÁNÍ – VLASTNOSTI

- expanduje vždy uzel, který se zdá nejblíže k cíli
- cesta nalezená v příkladu ($g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{Fagaras} \rightarrow \text{Bukurest}) = 450$) je sice úspěšná, ale není optimální ($g(\text{Arad} \rightarrow \text{Sibiu} \rightarrow \text{RimnicuVilcea} \rightarrow \text{Pitesti} \rightarrow \text{Bukurest}) = 418$)
- *úplnost* obecně není úplný (nekonečný prostor, cykly)

  optimálnost není optimální

  časová složitost $O(b^m)$, hodně záleží na h

  prostorová složitost $O(b^m)$, každý uzel v paměti

HLEDÁNÍ NEJLEPŠÍ CESTY – ALGORITMUS A*

- některé zdroje označují tuto variantu jako Best-first Search
- ohodnocovací funkce – kombinace $g(n)$ a $h(n)$:

$$f(n) = g(n) + h(n)$$

$g(n)$ je cena cesty do n

$h(n)$ je odhad ceny cesty z n do cíle

$f(n)$ je odhad ceny nejlevnější cesty, která vede přes n

- A* algoritmus vyžaduje tzv. přípustnou (*admissible*) heuristiku:

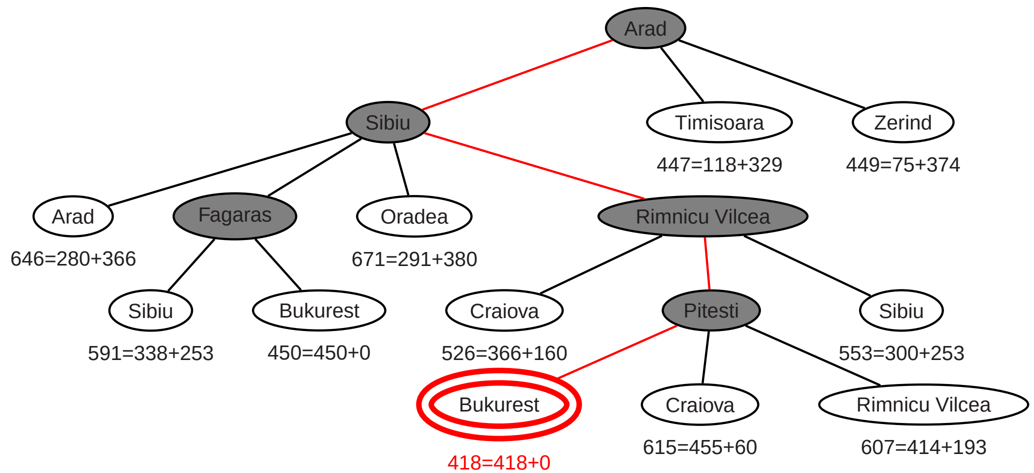
$$0 \leq h(n) \leq h^*(n), \text{ kde } h^*(n) \text{ je skutečná cena cesty z } n \text{ do cíle}$$

tj. odhad se volí vždycky kratší nebo roven ceně libovolné možné cesty do cíle
Např. přímá vzdálenost $h_{\text{vzd\_Buk}}$ nikdy není delší než (jakákoliv) cesta

HEURISTICKÉ HLEDÁNÍ A* – PŘÍKLAD

Hledání cesty z města *Arad* do města *Bukurest*

ohodnocovací funkce $f(n) = g(n) + h(n) = g(n) + h_{\text{vzd\_Buk}}(n)$, přímá vzdálenost z n do Bukuresti



Hledání nejlepší cesty – algoritmus A*

reprezentace uzlů:

- **l(N,F/G)** ... listový uzel **N**, **F** $= f(N) =$ **G** $+ h(N)$, **G** $= g(N)$
- **t(N,F/G,Subs)** ... podstrom s kořenovým uzlem **N**, **Subs** seznam podstromů seřazených podle f, **G** $= g(N)$ a **F** $=$ f-hodnota nejnadějnějšího následníka uzlu N

biggest(-Big) *horní závora pro cenu nejlepší cesty*

```
bestsearch(Start,Solution) :- biggest(Big), expand([],l(Start,0/0),Big,_,yes,Solution).

expand(P,l(N,_),_,_,yes,[N|P]) :- goal(N).
% list – generuj následníky a expanduj je v rámci Bound
expand(P,l(N,F/G),Bound,Tree1,Solved,Sol) :- F=<Bound,
    (bagof(M/C,(move(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).
% nelist , f<Bound – expanduj nejslibnější podstrom, pokračuj dle výsledku
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.
...
```

expand(+Path,+Tr,+Bnd,-Tr1,?Solved,-Sol)
Path – *cesta mezi kořenem a* **Tr**
Tr – *prohledávaný podstrom*
Bnd – f*-limita pro expandování* **Tr**
Tr1 – **Tr** *expandovaný až po* **Bnd**
Solved – **yes**, **no**, **never**
Sol – *cesta z kořene do cílového uzlu*

Hledání nejlepší cesty – algoritmus A* pokrač.

```
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).
```

continue(+Path, +Tree, +Bound, -NewTree, +SubtrSolved, ?TreeSolved, -Solution) *volba způsobu pokračování podle výsledků* **expand**

succlist(+G0, [+Node1/+Cost1, ...], [l(-BestNode,-BestF/-G), ...]) *setřídění seznamu listů podle f-hodnot*

vloží **T** *do seznamu stromů* **Ts** *podle* f

"vytáhne" **F** *ze struktury*

nejlepší f-hodnota ze seznamu stromů

Hledání nejlepší cesty A* – vlastnosti

- expanduje uzly podle $f(n) = g(n) + h(n)$

  A* expanduje všechny uzly s $f(n) < C^*$
  A* expanduje některé uzly s $f(n) = C^*$
  A* neexpanduje žádné uzly s $f(n) > C^*$

- *úplnost* je úplný (pokud $[$počet uzlů s $f < C^*] \neq \infty$)

  optimálnost je optimální

  časová složitost $O\big((b^*)^d\big)$, exponenciální v délce řešení d

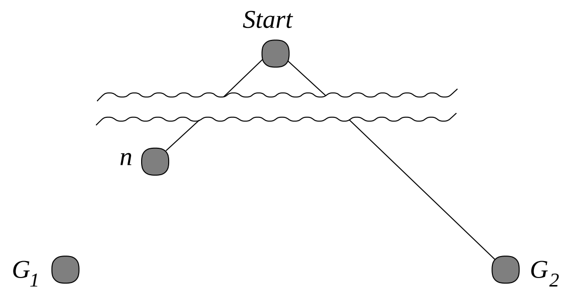
  b^* ... tzv. *efektivní faktor větvení*, viz dále

  prostorová složitost $O\big((b^*)^d\big)$, každý uzel v paměti

Problém s prostorovou složitostí řeší některé nedávné algoritmy (např. *Memory-bounded heuristic search*)

Důkaz optimálnosti algoritmu A*

- předpokládejme, že byl vygenerován nějaký suboptimální cíl G_2 a je uložen ve frontě.
- dále nechť n je neexpandovaný uzel na nejkratší cestě k optimálnímu cíli G_1 (tj. *chybně neexpandovaný* uzel ve správném řešení)



Pak

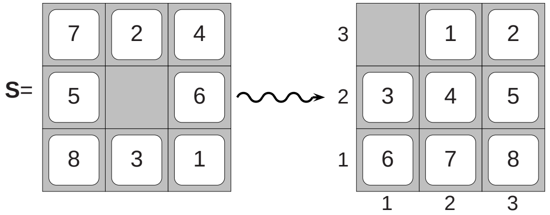
$$\begin{aligned} f(G_2) &= g(G_2) && \text{protože } h(G_2) = 0 \\ &> g(G_1) && \text{protože } G_2 \text{ je suboptimální} \\ &\geq f(n) && \text{protože } h \text{ je přípustná} \end{aligned}$$

tedy $f(G_2) > f(n)$ a $\Rightarrow$ A* nikdy nevybere G_2 pro expanzi dřív než expanduje n
$\longrightarrow$ spor s předpokladem, že n je *neexpandovaný uzel* □

Příklad – řešení posunovačky

konfigurace = seznam dvojic **X/Y** (sloupec/řádek) = [pozice$_{\text{díry}}$, pozice$_{\text{kámen č.1}}$, ...]

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



Volba přípustné heuristické funkce h:

➜ $h_1(n)$ = počet dlaždiček, které nejsou na svém místě $\quad h_1(\mathbf{S}) = 8$

➜ $h_2(n)$ = součet manhattanských vzdáleností dlaždic od svých správných pozic

$h_2(\mathbf{S}) = 3_7 + 1_2 + 2_4 + 2_5 + 3_6 + 2_8 + 2_3 + 3_1 = 18$

h_1 i h_2 jsou přípustné ... $h^*(S) = 26$

Určení kvality heuristiky

efektivní faktor větvení b^* – N... počet vygenerovaných uzlů, d... hloubka řešení:

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

např.: když A* najde řešení po 52 uzlech v hloubce 5 ... $b^* = 1.92$

heuristika je tím lepší, čím blíže je b^* hodnotě 1.

☞ měření b^* na malé množině testovacích sad – dobrá představa o přínosu heuristiky

	Průměrný počet uzlů			Efektivní faktor větvení b^*		
d	IDS	A$^*(h_1)$	A$^*(h_2)$	IDS	A$^*(h_1)$	A$^*(h_2)$
2	10	6	6	2.45	1.79	1.79
6	680	20	18	2.73	1.34	1.30
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
18	–	3056	363	–	1.46	1.26
24	–	39135	1641	–	1.48	1.26

h_2 dominuje h_1 $\big(\forall n : h_2(n) \geq h_1(n)\big)$ ... h_2 je lepší (nebo stejná) než h_1 ve všech případech

Jak najít přípustnou heuristickou funkci?

➜ je možné najít obecné pravidlo, jak objevit heuristiku h_1 nebo h_2?

➜ h_1 i h_2 jsou délky cest pro zjednodušené verze problému Posunovačka:

- při přenášení dlaždice kamkoliv – h_1=počet kroků nejkratšího řešení
- při posouvání dlaždice kamkoliv o 1 pole (i na plné) – h_2=počet kroků nejkratšího řešení

➜ relaxovaný problém – méně omezení na akce než původní problém

Cena optimálního řešení relaxovaného problému je přípustná heuristika pro původní problém.

optimální řešení původního problému = řešení relaxovaného problému

Posunovačka a relaxovaná posunovačka:

➜ dlaždice se může přesunout z A na B ⇔ A sousedí s B $\wedge$ B je prázdná.

➜ (a) dlaždice se může přesunout z A na B ⇔ A sousedí s B. .... h_2

(b) dlaždice se může přesunout z A na B ⇔ B je prázdná. .... Gaschnigova heuristika

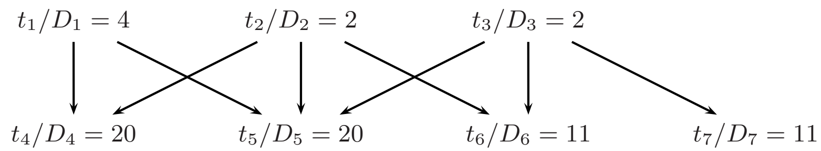
(c) dlaždice se může přesunout z A na B. .... h_1

Příklad – rozvrh práce procesorů

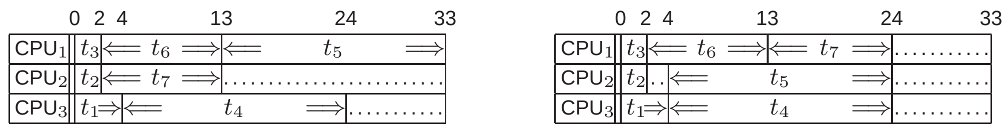
➜ úlohy t_i s potřebným časem na zpracování D_i (např.: $i = 1, \ldots, 7$)

➜ m procesorů (např.: $m = 3$)

➜ relace precedence mezi úlohami – které úlohy mohou začít až po skončení dané úlohy



➜ problém: najít rozvrh práce pro každý procesor s minimalizací celkového času



PŘÍKLAD – ROZVRH PRÁCE PROCESORŮ pokrač.

- stavy: **nezařazené_úlohy*zařazené_úlohy*čas_ukončení**

  např.: **[WaitingTask1/D1,WaitingTask2/D2,...]*[Task1/F1,Task2/F2,...]*FinTime**

- přechodová funkce **move(+Uzel, -NaslUzel, -Cena)**:

```
move(Tasks1*[_/F|Active]*Fin1, Tasks2*Active2*Fin2, Cost) :-
    del1(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.
move(Tasks*[_/F|Active1]*Fin,Tasks*Active2*Fin,0) :- insertidle(F,Active1,Active2).

before(T1,T2) :- precedence(T1,T2).
before(T1,T2) :- precedence(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).

goal([]*_*_).
```


PŘÍKLAD – ROZVRH PRÁCE PROCESORŮ pokrač.

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

optimální (nedosažitelný) čas:

$$\textbf{Finall} = \frac{\sum_i D_i + \sum_j F_j}{m}$$

skutečný čas výpočtu: $\textbf{Fin} = \max(F_j)$

heuristická funkce h:

$$\textbf{H} = \begin{cases} \textbf{Finall} - \textbf{Fin}, & \text{když } \textbf{Finall} > \textbf{Fin} \\ 0, & \text{jinak} \end{cases}$$

```
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.

precedence(t1, t4). precedence(t1, t5).
...
```

Dekompozice problému, AND/OR grafy

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

➜ Připomínka – průběžná písemka

➜ Příklad – Hanoiské věže

➜ AND/OR grafy

➜ Prohledávání AND/OR grafů

PŘIPOMÍNKA – PRŮBĚŽNÁ PÍSEMKA

➜ termín – příští týden, 25. října, 14:00, B204, na přednášce

➜ náhradní termín: není

➜ příklady (formou testu – odpovědi A, B, C, D, E, z látky probrané do 25.10.):

- uveden příklad v Prologu, otázka Co řeší tento program?
- uveden příklad v Prologu a cíl, otázka Co je (návratová) hodnota výsledku?
- upravte (doplňte/zmeňte řádek) uvedený program tak, aby. . .
- uvedeno několik tvrzení, potvrďte jejich pravdivost/nepravdivost
- porovnání vlastností několika algoritmů

➜ rozsah: 4 příklady

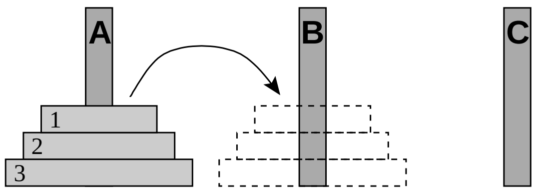
➜ hodnocení: max. 32 bodů – za *správnou odpověď* 8 bodů, za *žádnou odpověď* 0 bodů, za *špatnou odpověď* -3 bodů.

PŘÍKLAD – HANOISKÉ VĚŽE

➜ máme tři tyče: **A**, **B** a **C**.

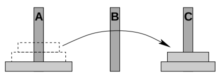
➜ na tyči **A** je (podle velikosti) n kotoučů.

➜ úkol: přeskládat z **A** pomocí **C** na tyč **B** (zaps. n**(A, B, C)**) bez porušení uspořádání

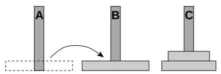


Můžeme rozložit na fáze:

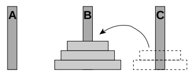
1. přeskládat $n-1$ kotoučů z **A** pomocí **B** na **C**.



2. přeložit 1 kotouč z **A** na **B**

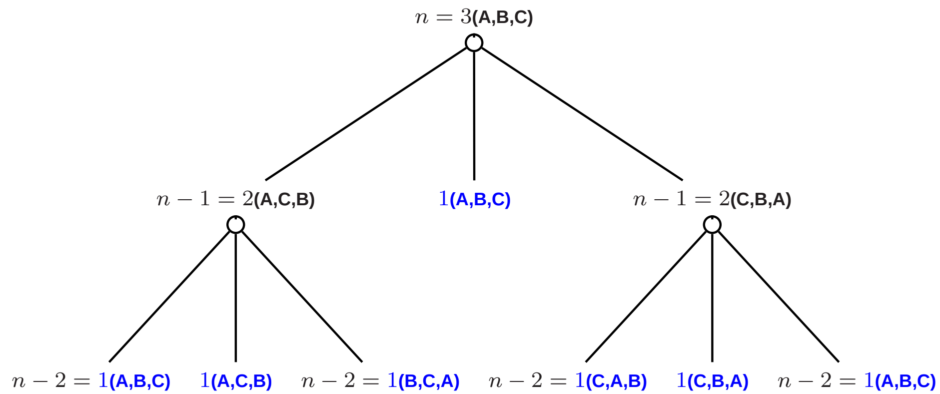


3. přeskládat $n-1$ kotoučů z **C** pomocí **A** na **B**



PŘÍKLAD – HANOISKÉ VĚŽE pokrač.

schéma celého řešení pro $n = 3$:



Příklad – Hanoiské věže pokrač.

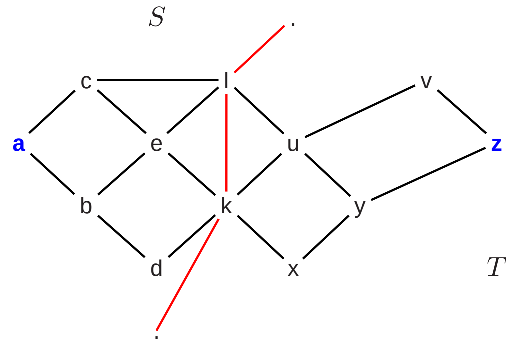
```
?−op(100,xfx,to), dynamic(hanoi/5).

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 :− !)).

?− hanoi(3,a,b,c,M).
    M = [a to b, a to c, b to c, a to b, c to a, c to b, a to b] ;
    No
```


Cesta mezi městy pomocí AND/OR grafů

města: **a**, . . . , **e** . . . ve státě S

l a **k** . . . hraniční přechody

u, . . . , **z** . . . ve státě T



hledáme cestu z **a** do **z**:

➜ cesta z **a** do hraničního přechodu

➜ cesta z hraničního přechodu do **z**

Cesta mezi městy pomocí AND/OR grafů pokrač.

schéma řešení pomocí rozkladu na podproblémy = AND/OR graf

přímý zápis AND/OR grafu v Prologu:

OR uzel **v** s následníky **u1**, **u2**, . . . , **uN**:

```
v :− u1.
v :− u2.
...
v :− uN.
```

AND uzel **x** s následníky **y1**, **y2**, . . . , **yM**:

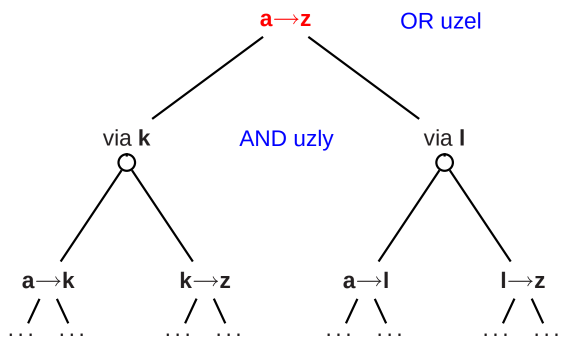
```
x :− y1, y2, ..., yM.
```

cílový uzel **g** ($\triangleq$ elementární problém):

```
g.
```

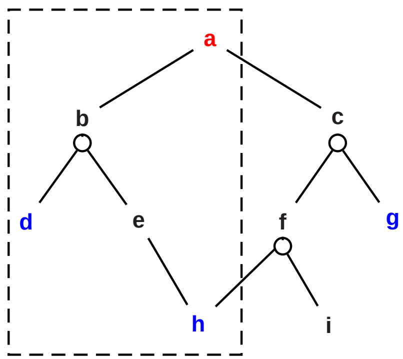
kořenový uzel **root**:

```
?− root.
```



Celkové řešení = podgraf AND/OR grafu, který nevynechává žádného následníka AND-uzlu.

Triviální prohledávání AND/OR grafu v Prologu




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

?− a.
    Yes
```

Reprezentace AND/OR grafu

AND/OR graf = graf s 2 typy vnitřních uzlů – AND uzly a OR uzly

- *AND uzel* jako součást řešení vyžaduje průchod všech svých poduzlů
- *OR uzel* se chová jako běžný uzel klasického grafu

Reprezentace AND/OR grafu v Prologu:

- zavedeme operátory '--->' a ':'

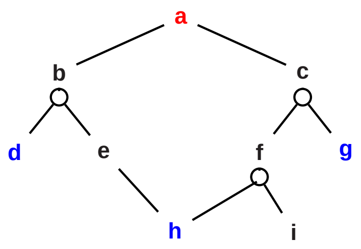
```
?- op(600, xfx, --->).
?- op(500, xfx, :).
```

op(+Priorita, +Typ, +Jméno)

Priorita *číslo 0..1200*
Typ *jedno z* **xf**, **yf**, **xfx**, **xfy**, **yfx**, **yfy**, **fy** *nebo* **fx**
Jméno *funktor nebo symbol*

- AND/OR graf budeme zapisovat

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

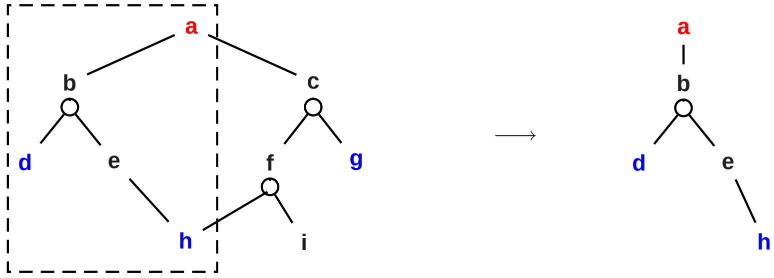



```
a ---> or:[b,c].
b ---> and:[d,e].
c ---> and:[f,g].
e ---> or:[h].
f ---> and:[h,i].
goal(d).
goal(g).
goal(h).
```


Strom řešení AND/OR grafu

strom řešení T problému P s AND/OR grafem G:

- problém P je kořen stromu T
- jestliže P je OR uzel grafu $G \Rightarrow$ právě jeden z jeho následníků se svým stromem řešení je v T
- jestliže P je AND uzel grafu $G \Rightarrow$ všichni jeho následníci se svými stromy řešení jsou v T
- každý list stromu řešení T je cílovým uzlem v G



Prohledávání AND/OR grafu do hloubky

```
% solve(Node, SolutionTree)
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([Node1,Node2, ...], [SolutionTree1, SolutionTree2, ...])
solveall([],[]).
solveall([Node|Nodes],[Tree|Trees]) :- solve(Node,Tree), solveall(Nodes,Trees).

?- solve(a,Tree).
    Tree = a---> (b--->and:[d, e--->h]) ;
    No
```


Heuristické prohledávání AND/OR grafu

- doplnění reprezentace o cenu přechodové hrany (=odhad složitosti podproblému):

```
Uzel ---> AndOr:[NaslUzel1/Cena1, NaslUzel2/Cena2, ..., NaslUzelN/CenaN].
```

- definujeme cenu uzlu jako cenu optimálního řešení jeho podstromu
- pro každý uzel N máme daný odhad jeho ceny:

$$h(N) = \text{heuristický odhad ceny optimálního podgrafu s kořenem } N$$

- pro každý uzel N, jeho následníky $N_1, \ldots, N_b$ a jeho předchůdce M definujeme:

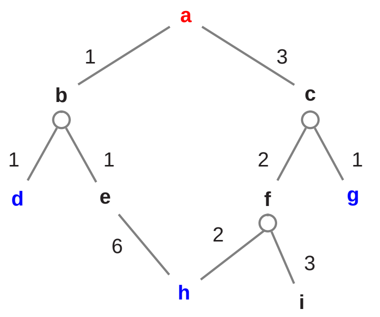
$$F(N) = \begin{cases} h(N), & \text{pro ještě neexpandovaný uzel } N \\ 0, & \text{pro cílový uzel (elementární problém)} \\ \text{cena}(M,N) + \min_i(F(N_i)), & \text{pro OR-uzel } N \\ \text{cena}(M,N) + \sum_i F(N_i), & \text{pro AND-uzel } N \end{cases}$$

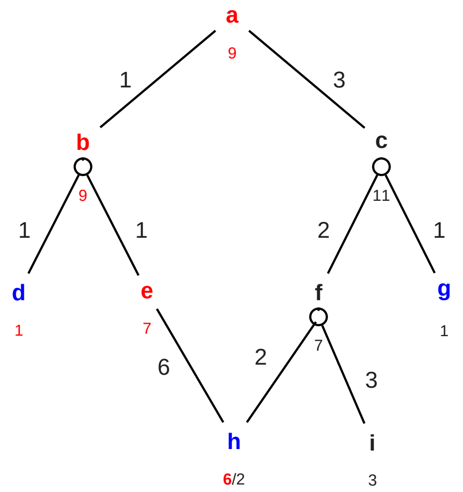
Heuristické prohledávání AND/OR grafu – příklad

setříděný seznam částečně expandovaných grafů =

[Nevyřešený$_1$, Nevyřešený$_2$, ..., Vyřešený$_1$, ...]

$F_{\text{Nevyřešený}_1} \leq F_{\text{Nevyřešený}_2} \leq \ldots$





Reprezentace AND/OR grafu při heuristickém prohledávání

C ... cena hrany do uzlu N

F ... příslušná heuristická F-hodnota uzlu N

N ... identifikátor uzlu

list AND/OR grafu ... struktura **leaf(N,F,C)**.

$\mathbf{F} = \mathbf{C} + h(\mathbf{N})$

OR uzel AND/OR grafu ... struktura **tree(N,F,C,or:[T1,T2,T3,...])**.

$\mathbf{F} = \mathbf{C} + \min_i \mathbf{F}_i$

AND uzel AND/OR grafu ... struktura **tree(N,F,C,and:[T1,T2,T3,...])**.

$\mathbf{F} = \mathbf{C} + \sum_i \mathbf{F}_i$

vyřešený list AND/OR grafu ... struktura **solvedleaf(N,F)**.

$\mathbf{F} = \mathbf{C}$

vyřešený OR uzel AND/OR grafu ... struktura **solvedtree(N,F,T)**.

$\mathbf{F} = \mathbf{C} + \mathbf{F}_1$

vyřešený AND uzel AND/OR grafu ... struktura **solvedtree(N,F,and:[T1,T2,...])**.

$\mathbf{F} = \mathbf{C} + \sum_i \mathbf{F}_i$

Heuristické prohledávání AND/OR grafu

```
andor(Node,SolutionTree) :- biggest(Bound),expand(leaf(Node,0,0),Bound,SolutionTree,yes).

% Case 1: bound exceeded, in all remaining cases F =< Bound
expand(Tree,Bound,Tree,no) :- f(Tree,F),F>Bound,!.
% Case 2: goal encountered
expand(leaf(Node,F,C),_,solvedleaf(Node,F),yes) :- goal(Node),!.
% Case 3: expanding a leaf
expand(leaf(Node,F,C),Bound,NewTree,Solved) :- expandnode(Node,C,Tree1),!,
    (expand(Tree1,Bound,NewTree,Solved);Solved=never,!).
% Case 4: expanding a tree
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).
```

expand(+Tree, +Bound, -NewTree, ?Solved) *expanduje* **Tree** *po* **Bound**. *Výsledek je* **NewTree** *se stavem* **Solved**

expandlist *všechny grafy v seznamu* **Trees** *se závorou* **Bound**. *Výsledek je v seznamu* **NewTrees** *a celkový stav v* **Solved**

continue *určuje, jak pokračovat po expanzi seznamu grafů*

Heuristické prohledávání AND/OR grafu pokrač.

```
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,or: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,Trees1),
    insert(leaf(Node,F,C),Trees1,Trees).

allsolved([]).
allsolved([Tree|Trees]) :- solved(Tree),allsolved(Trees).

solved(solvedtree(_,_,_)).
solved(solvedleaf(_,_)).
```

combine *kombinuje výsledky expanze stromu a seznamu stromů*

expandnode *vyrobí z uzlu a jeho následovníků strom*

allsolved *zkontroluje, jestli všechny stromy v seznamu jsou vyřešené*

Heuristické prohledávání AND/OR grafu pokrač.

```
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).

% First tree in OR−list is best
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).

% The only candidate
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).
```

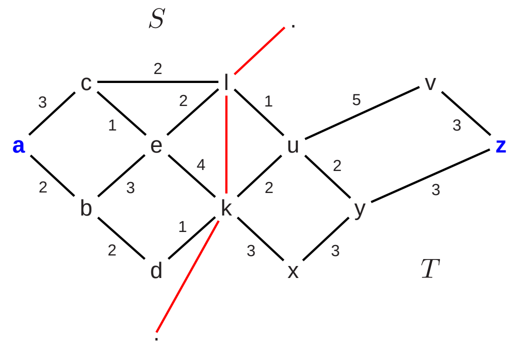
insert *vkládá strom do seznamu stromů se zachováním třídění*

backup *vyhledá uloženou F-hodnotu AND/OR stromu/uzlu*

selecttree(+Trees, -BestTree, -OtherTrees, +Bound, -Bound1) *vybere* **BestTree** *z* **Trees**, *zbytek je v* **OtherTrees**. **Bound** *je závora pro* **Trees**, **Bound1** *pro* **BestTree**

Cesta mezi městy heuristickým AND/OR hledáním

- ➜ cesta mezi **Mesto1** a **Mesto2** – predikát **move(Mesto1,Mesto2,Vzdal)**.
- ➜ klíčové postavení města **Mesto3** – predikát **key(Mesto1–Mesto2,Mesto3)**.




```
move(a,b,2).  move(a,c,3).  move(b,e,3).
move(b,d,2).  move(c,e,1).  move(c,l,2).
move(e,k,4).  move(e,l,2).  move(k,u,2).
move(k,x,3).  move(u,v,5).  move(x,y,3).
move(y,z,3).  move(v,z,3).  move(l,u,1).
move(d,k,1).  move(u,y,2).

stateS(a).  stateS(b).  stateS(c).  stateS(d).  stateS(e).
stateT(u).  stateT(v).  stateT(x).  stateT(y).  stateT(z).
border(l).  border(k).

key(M1−M2,M3) :− stateS(M1), stateT(M2), border(M3).

city(X) :− (stateS(X);stateT(X);border(X)).
```


Cesta mezi městy heuristickým AND/OR hledáním pokrač.

vlastní hledání cesty:

1. **Y1**, **Y2**,... klíčové body mezi městy **A** a **Z**. 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 ⇒ hledej souseda **Y** města **A** takového, že existuje cesta z **Y** do **Z**.

Cesta mezi městy heuristickým AND/OR hledáním pokrač.

Konstrukce příslušného AND/OR grafu:

```
?− op(560,xfx,via).   % operátory X−Z a X−Z via Y

a−z −−−> or:[a−z via k/0,a−z via l/0]
a−v −−−> or:[a−v via k/0,a−v via l/0]
 ...
a−l −−−> or:[c−l/3,b−l/2]
b−l −−−> or:[e−l/3,d−l/2]
 ...
a−z via l −−−> and:[a−l/0,l−z/0]
a−v via l −−−> and:[a−l/0,l−v/0]
 ...
goal(a−a). goal(b−b).  ...
```

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

Když $\forall n : h(n) \leq h^*(n)$, kde h^* je minimální cena řešení uzlu n ⇒ najdeme vždy optimální řešení

Problémy s omezujícími podmínkami

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

- ➜ Průběžná písemná práce
- ➜ Problémy s omezujícími podmínkami
- ➜ CLP – Constraint Logic Programming
- ➜ Příklad – algebrogram
- ➜ Řešení problémů s omezujícími podmínkami
- ➜ Příklad – problém N dam

PRŮBĚŽNÁ PÍSEMNÁ PRÁCE

- ➜ délka pro vypracování: 25 minut
- ➜ nejsou povoleny žádné materiály
- ➜ u odpovědí typu A, B, C, D, E:
  - pouze jedna odpověď je nejsprávnější ☺
  - za tuto nejsprávnější je 8 bodů
  - za žádnou odpověď je 0 bodů
  - za libovolnou jinou, případně za nejasné označení odpovědi je mínus 3 body
- ➜ celkové hodnocení 0 až 32 bodů (celkové záporné hodnocení se bere jako 0)

PROBLÉMY S OMEZUJÍCÍMI PODMÍNKAMI

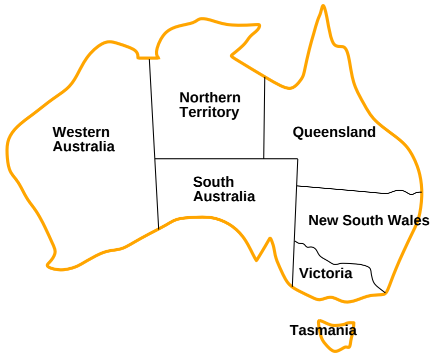
- ➜ standardní problém řešený prohledáváním stavového prostoru $\longrightarrow$ stav je "*černá skříňka*" – pouze cílová podmínka a přechodová funkce
- ➜ problém s omezujícími podmínkami, *Constraint Satisfaction Problem*, CSP:
  - n-tice proměnných $X_1, X_2, \ldots, X_n$ s hodnotami z domén $D_1, D_2, \ldots, D_n$, $D_i \neq \emptyset$
  - množina omezení $C_1, C_2, \ldots, C_m$ nad proměnnými X_i
  - stav = přiřazení hodnot proměnným $\{X_i = v_i, X_j = v_j, \ldots\}$

    konzistentní přiřazení neporušuje žádné z omezení C_i

    úplné přiřazení zmiňuje každou proměnnou X_i
  - řešení = úplné konzistentní přiřazení hodnot proměnným

    někdy je ještě potřeba maximalizovat *cílovou funkci*
- ➜ výhody:
  - jednoduchý formální jazyk pro specifikaci problému
  - může využívat obecné heuristiky (ne jen specifické pro daný problém)

PŘÍKLAD – OBARVENÍ MAPY



Proměnné WA, NT, Q, NSW, V, SA, T

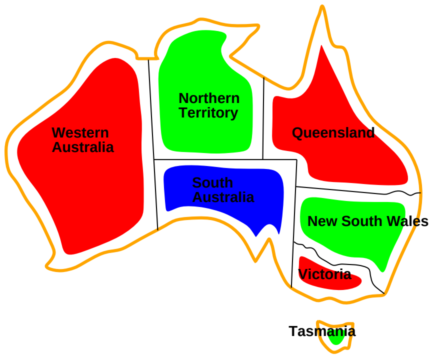
Domény $D_i = \{$červená, zelená, modrá$\}$

Omezení – sousedící oblasti musí mít různou barvu

tj. pro každé dvě sousedící: $WA \neq NT$ nebo

$(WA, NT) \in \{($červená, zelená$), ($červená, modrá$), ($zelená, modrá$), \ldots\}$

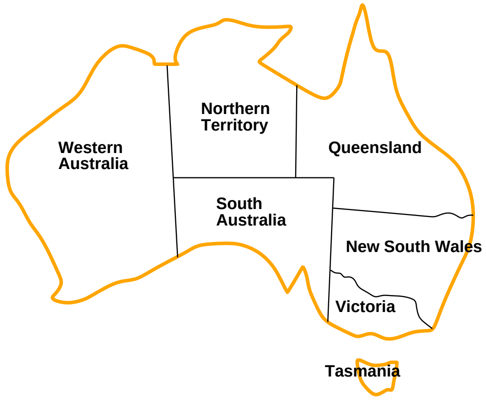
PŘÍKLAD – OBARVENÍ MAPY pokrač.

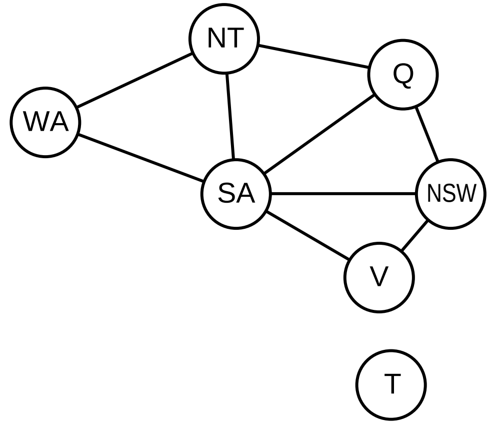


Řešení – konzistentní přiřazení všem proměnným:
$\{WA =$ červená, $NT =$ zelená, $Q =$ červená, $NSW =$ zelená, $V =$ červená, $SA =$ modrá, $T =$ zelená$\}$

GRAF OMEZENÍ

Pro binární omezení: uzly = proměnné, hrany = reprezentují jednotlivá omezení





Algoritmy pro řešení CSP využívají této grafové reprezentace omezení

VARIANTY CSP PODLE HODNOT PROMĚNNÝCH

- diskrétní hodnoty proměnných – každá proměnná má jednu konkrétní hodnotu
  - konečné domény
    - např. Booleovské (včetně NP-úplných problémů splnitelnosti)
    - výčtové
  - nekonečné domény – čísla, řetězce, . . .
    - např. rozvrh prací – proměnné = počáteční/koncový den každého úkolu
    - vyžaduje jazyk omezení, např. $StartJob_1 + 5 \leq StartJob_3$
    - číselné *lineární* problémy jsou řešitelné, *nelineární* obecné řešení nemají
- spojité hodnoty proměnných
  - časté u reálných problémů
  - např. počáteční/koncový čas měření na Hubbleově teleskopu (závisí na astronomických, precedenčních a technických omezeních)
  - *lineární omezení* řešené pomocí Lineárního programování (omezení = lineární nerovnice tvořící konvexní oblast) $\longrightarrow$ jsou řešitelné v polynomiálním čase

VARIANTY OMEZENÍ

- unární omezení zahrnuje jedinou proměnnou
  např. $SA \neq$ zelená
- binární omezení zahrnují dvě proměnné
  např. $SA \neq WA$
- omezení vyššího řádu zahrnují 3 a více proměnných
  např. kryptoaritmetické omezení na sloupce u algebrogramu
- preferenční omezení (soft constraints), např. 'červená je lepší než zelená'
  možno reprezentovat pomocí ceny přiřazení u konkrétní hodnoty a konkrétní proměnné $\longrightarrow$ hledá se optimalizované řešení vzhledem k ceně

CLP – Constraint Logic Programming

```
% SICStus Prolog
:- use_module(library(clpfd)).    % clpq, clpr

?- X in 1..5, Y in 2..8, X+Y #= T.
     X in 1..5,
     Y in 2..8,
     T in 3..13
     Yes

?- X in 1..5, Y in 2..8, X+Y #= T, labeling([],[X,Y,T]).
     T = 3,
     X = 1,
     Y = 2
     Yes
```

aritmetická omezení ...
- ➜ *rel. operátory #=, #\=, #<, #=<, #>, #>=*
- ➜ **sum(Variables,RelOp,Suma)**

výroková omezení ...
- *\# negace, #/\ konjunkce,*
- *#\/ disjunkce, #<=> ekvivalence*

domain(+Variables, +Min, +Max)
?X in +Min..+Max
?X in +Range ...
A in (1..3) \/(8..15) \/(5..9) \/100.
fd_dom(?Var,?Range) *zjištění domény proměnné*
fd_set(?Var,?FDSet), ?X in_set +FDSet *příslušnost k dané konečné doméně*

CLP – Constraint Logic Programming pokrač.

```
?- X #< 4, domain([X,Y],0,5).
     X in 0..3, Y in 0..5 ?
     Yes

?- X #< 4, indomain(X).
     Instantiation error

?- X #> 3, X #< 6, indomain(X).
     X = 4 ? ;
     X = 5 ? ;
     No

?- X in 4..sup, X #\= 17, fd_set(X,F).
     F = [[4|16],[18|sup]],
     X in (4..16) \/ (18..sup) ?
     Yes
```


Příklad – algebrogram

```
    S E N D
+   M O R E
-----------
  M O N E Y
```

Proměnné $\{S, E, N, D, M, O, R, Y\}$

Domény $D_i = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Omezení
- $S > 0, M > 0$
- $S \neq E \neq N \neq D \neq M \neq O \neq R \neq Y$
- $D + E = R + 10 * X_1,$
  $N + R + X_1 = E + 10 * X_2, \ldots$

```
moremoney([S,E,N,D,M,O,R,Y], Type) :- domain([S,E,N,D,M,O,R,Y],0,9),
                                      S #> 0, M #> 0,
                                      all_different ([S,E,N,D,M,O,R,Y]),
                                      sum(S,E,N,D,M,O,R,Y),
                                      labeling(Type, [S,E,N,D,M,O,R,Y]).

sum(S,E,N,D,M,O,R,Y):-          1000*S + 100*E + 10*N + D
                        +       1000*M + 100*O + 10*R + E
                        #= 10000*M + 1000*O + 100*N + 10*E + Y.

?-moremoney([S,E,N,D,M,O,R,Y],[]). % Type=[] ... Type = [ leftmost , step , up, all ]
D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2 ?
Yes
```


Inkrementální formulace CSP

CSP je možné převést na standardní prohledávání takto:

- **stav** – přiřazení hodnot proměnným
- **počáteční stav** – prázdné přiřazení $\{\}$
- **přechodová funkce** – přiřazení hodnoty libovolné dosud nenastavené proměnné tak, aby výsledné přiřazení bylo konzistentní
- **cílová podmínka** – aktuální přiřazení je úplné
- **cena cesty** – konstantní (např. 1) pro každý krok

1. platí beze změny pro všechny CSP!
2. prohledávací strom dosahuje hloubky n (počet proměnných) a řešení se nachází v této hloubce $(d = n) \Rightarrow$ je vhodné použít prohledávání do hloubky

PROHLEDÁVÁNÍ S NAVRACENÍM

- ➜ přiřazení proměnným jsou komutativní
  tj. $[1.\ WA =$ červená, $\quad 2.\ NT =$ zelená$]$ je totéž jako $[1.\ NT =$ zelená, $\quad 2.\ WA =$ červená$]$
- ➜ stačí uvažovat pouze přiřazení jediné proměnné v každém kroku $\Rightarrow$ počet listů d^n
- ➜ prohledávání do hloubky pro CSP – tzv. prohledávání s navracením (*backtracking search*)
- ➜ prohledávání s navracením je základní neinformovaná strategie pro řešení problémů s omezujícími podmínkami
- ➜ schopný vyřešit např. problém n-dam pro $n \approx 25$

PŘÍKLAD – PROBLÉM N DAM

```
queens(N,L,Type):- length(L,N),
                   domain(L,1,N),
                   constr_all (L),
                   labeling(Type,L).

constr_all ([]).
constr_all ([X|Xs]):- constr_between(X,Xs,1), constr_all(Xs).

constr_between(_,[], _).
constr_between(X,[Y|Ys],N):-
    no_threat(X,Y,N),
    N1 is N+1,
    constr_between(X,Ys,N1).

no_threat(X,Y,J):- X #\= Y, X+J #\= Y, X-J #\= Y.

?- queens(4, L, [ff ]).
    L = [2,4,1,3] ? ;
    L = [3,1,4,2] ? ;
    No
```

1. definice proměnných a domén

2. definice omezení

3. hledání řešení

OVLIVNĚNÍ EFEKTIVITY PROHLEDÁVÁNÍ S NAVRACENÍM

Obecné metody ovlivnění efektivity:

- Která proměnná dostane hodnotu v tomto kroku?
- V jakém pořadí zkoušet přiřazení hodnot konkrétní proměnné?
- Můžeme předčasně detekovat nutný neúspěch v dalších krocích?

používané strategie:

- ❑ **nejomezenější proměnná** $\longrightarrow$ vybrat proměnnou s nejméně možnými hodnotami
- ❑ **nejvíce omezující proměnná** $\longrightarrow$ vybrat proměnnou s nejvíce omezeními na zbývající proměnné
- ❑ **nejméně omezující hodnota** $\longrightarrow$ pro danou proměnnou – hodnota, která zruší nejmíň hodnot zbývajících proměnných
- ❑ **dopředná kontrola** $\longrightarrow$ udržovat seznam možných hodnot pro zbývající proměnné
- ❑ **propagace omezení** $\longrightarrow$ navíc kontrolovat možné nekonzistence mezi zbývajícími proměnnými

OVLIVNĚNÍ EFEKTIVITY V CLP

V Prologu (CLP) možnosti ovlivnění efektivity – **labeling(Typ, ...)**:

```
?- constraints(Vars,Cost),
   labeling([ ff , bisect , down,minimize(Cost)],Vars).
```

- ❑ **výběr proměnné** – **leftmost**, **min**, **max**, **ff**, . . .
- ❑ **dělení domény** – **step**, **enum**, **bisect**, **value(Enum)**
- ❑ **prohledávání domény** – **up**, **down**
- ❑ **která řešení** – **all**, **minimize(X)**, **maximize(X)**, . . .

Hry a základní herní strategie

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

➜ Statistické výsledky průběžné písemky

➜ Hry vs. Prohledávání stavového prostoru

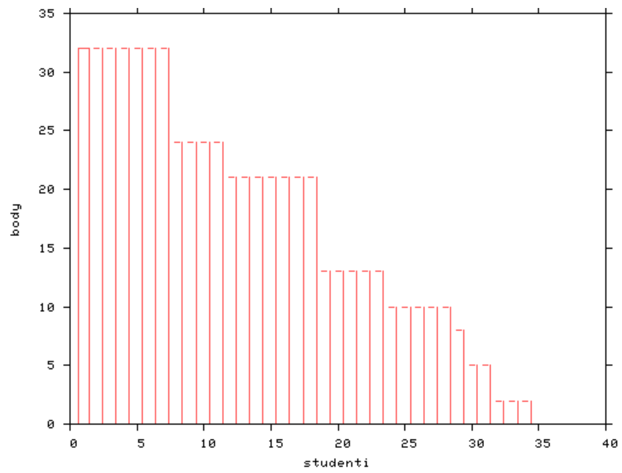
➜ Algoritmus Minimax

➜ Algoritmus Alfa-Beta prořezávání

➜ Nedeterministické hry

➜ Hry s nepřesnými znalostmi

Statistické výsledky průběžné písemky



průběžná písemka PB016

46 studentů

Body	Počet studentů
32	7
24	4
21	7
13	5
10	5
8	1
5	2
2	3
0	5

Hry × Prohledávání stavového prostoru

Multiagentní prostředí:

➜ agent musí brát v úvahu akce jiných agentů ⟶ jak ovlivní jeho vlastní prospěch

➜ vliv ostatních agentů – prvek náhody

➜ kooperativní × soupeřící multiagentní prostředí (MP)

Hry:

➜ matematická teorie her (odvětví ekonomie) – kooperativní i soupeřící MP, kde vliv všech agentů je *významný*

➜ hra v UI = obv. deterministické MP, 2 střídající se agenti, výsledek hry je vzájemně opačný nebo shoda

Algoritmy soupeřícího prohledávání (*adversarial search*):

➜ oponent dělá dopředu neurčitelné tahy ⟶ řešením je strategie, která počítá se všemi možnými tahy protivníka

➜ časový limit ⇒ zřejmě nenajdeme optimální řešení ⟶ hledáme lokálně optimální řešení

Hry a UI – historie

➜ Babbage, 1846 – počítač porovnává přínos různých herních tahů

➜ von Neumann, 1944 – algoritmy perfektní hry

➜ Zuse, Wiener, Shannon, 1945–50 – přibližné vyhodnocování

➜ Turing, 1951 – první šachový program (jen na papíře)

➜ Samuel, 1952–57 – strojové učení pro zpřesnění vyhodnocování

➜ McCarthy, 1956 – prořezávání pro možnost hlubšího prohledávání

řešení her je zajímavým předmětem studia ⟵ je obtížné:

průměrný faktor větvení v šachách $b = 35$

pro 50 tahů 2 hráčů … prohledávací strom $\approx 35^{100} \approx 10^{154}$ uzlů ($\approx 10^{40}$ stavů)

Hry a UI – aktuální výsledky

- **dáma** – 1994 program *Chinook* porazil světovou šampionku Marion Tinsley. Používá úplnou databázi tahů pro ≤ 8 figur (443 748 401 247 pozic).
- **šachy** – 1997 porazil stroj *Deep Blue* světového šampiona Gary Kasparova. Stroj počítá 200 mil pozic/s, sofistikované vyhodnocování a nezveřejněné metody pro prozkoumávání některých tahů až do hloubky 40 tahů.
- **Othello** – světoví šampioni odmítají hrát s počítači, protože stroje jsou příliš dobré
- **Go** – světoví šampioni odmítají hrát s počítači, protože stroje jsou příliš slabé. V Go je $b > 300$, takže počítače mohou používat pouze znalostní bázi vzorových her.

Typy her

	deterministické	*s náhodou*
perfektní znalosti	šachy, dáma, Go, Othello	backgammon, monopoly
nepřesné znalosti		bridge, poker, scrabble

Hledání optimálního tahu

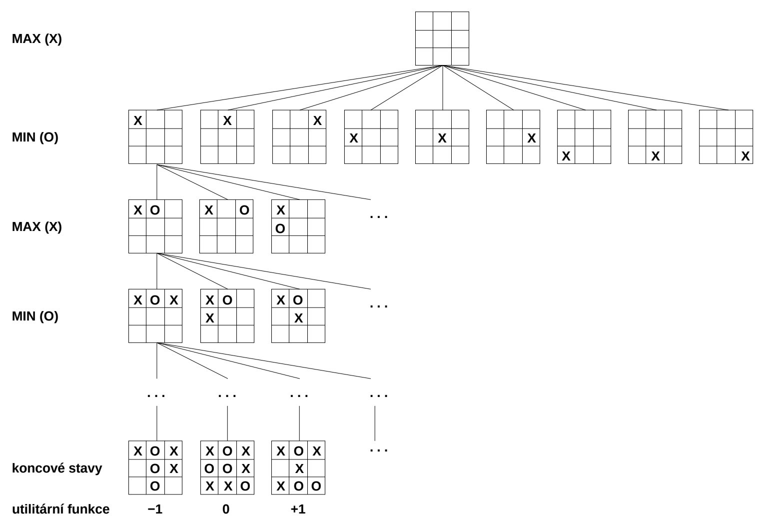
2 hráči – MAX a MIN, MAX je první na tahu a pak se střídají až do konce hry

hra = prohledávací problém:

- **počáteční stav** – počáteční herní situace + kdo je na tahu
- **přechodová funkce** – vrací dvojice $($legální tah, výsledný stav$)$
- **ukončovací podmínka** – určuje, kdy hra končí, označuje koncové stavy
- **utilitární funkce** – numerické ohodnocení koncových stavů

Hledání optimálního tahu pokrač.

počáteční stav a přechodová funkce definují herní strom:



Algoritmus Minimax

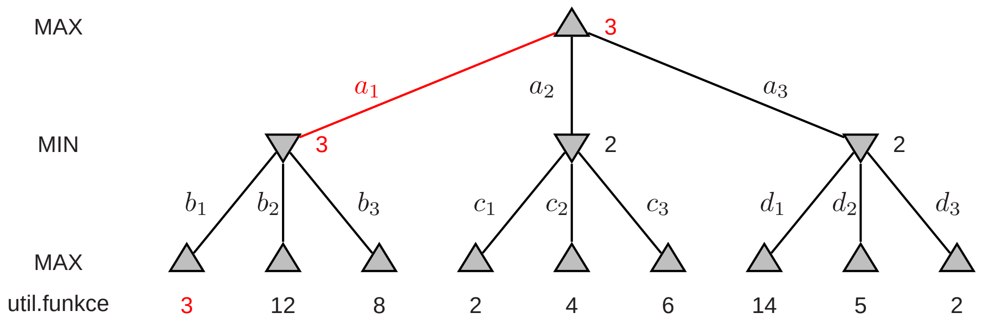
MAX (△) musí *prohledat* herní strom pro zjištění nejlepšího tahu proti MIN (▽)

$\longrightarrow$ zjistit nejlepší hodnotu minimax – zajišťuje *nejlepší výsledek* proti *nejlepšímu protivníkovi*

$$\text{Hodnota minimax}(n) = \begin{cases} \text{utility}(n) & \text{pro koncový stav } n \\ \max_{s \in \text{moves}(n)} \text{Hodnota minimax}(s) & \text{pro MAX uzel } n \\ \min_{s \in \text{moves}(n)} \text{Hodnota minimax}(s) & \text{pro MIN uzel } n \end{cases}$$

Algoritmus Minimax pokrač.

příklad – hra jen na jedno kolo = 2 tahy (půlkola)



Algoritmus Minimax pokrač.

```
% minimax( Pos, BestSucc, Val ):
%    Pos is a position , Val is its minimax value;
%    best move from Pos leads to position BestSucc
minimax( Pos, BestSucc, Val) :-
     moves( Pos, PosList), !,                 % Legal moves in Pos produce PosList
     best( PosList, BestSucc, Val)
     ;
     staticval ( Pos, Val).                   % Pos has no successors : evaluate statically

best( [ Pos], Pos, Val) :-
     minimax( Pos, _, Val), !.
best( [Pos1 | PosList], BestPos, BestVal) :-
     minimax( Pos1, _, Val1),
     best( PosList, Pos2, Val2),
     betterof( Pos1, Val1, Pos2, Val2, BestPos, BestVal).
betterof( Pos0, Val0, Pos1, Val1, Pos0, Val0) :-          % Pos0 better than Pos1
     min_to_move( Pos0),                                  % MIN to move in Pos0
     Val0 > Val1, !                                       % MAX prefers the greater value
     ;
     max_to_move( Pos0),                                  % MAX to move in Pos0
     Val0 < Val1, !.                              % MIN prefers the lesser value
betterof( Pos0, Val0, Pos1, Val1, Pos1, Val1).            % Otherwise Pos1 better than Pos0
```


Algoritmus Minimax – vlastnosti

úplnost	úplný pouze pro konečné stromy
optimálnost	je optimální proti optimálnímu oponentovi
časová složitost	$O(b^m)$
prostorová složitost	$O(bm)$, prohledávání do hloubky

šachy … $b \approx 35, m \approx 100 \Rightarrow$ přesné řešení není možné

$b^m = 10^6, b = 35 \Rightarrow m \approx 4$

4-tahy $\approx$ člověk-nováček

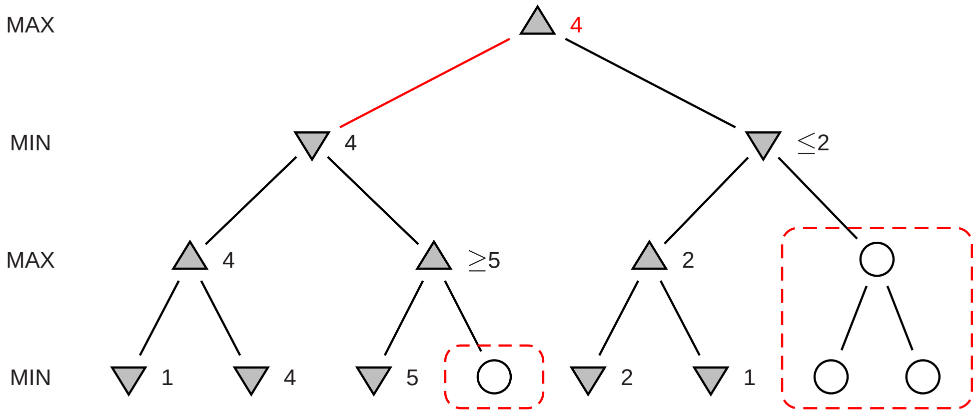
8-tahů $\approx$ člověk-mistr, typické PC

12-tahů $\approx$ Deep Blue, Kasparov

Algoritmus Alfa-Beta prořezávání

Příklad stromu, který zpracuje predikát **minmax**

Alfa-Beta odřízne expanzi některý uzlů ⇒ Alfa-Beta procedura je efektivnější variantou minimaxu



Algoritmus Alfa-Beta – vlastnosti

- ➜ prořezávání neovlivní výsledek ⇒ je stejný jako u minimaxu
- ➜ dobré uspořádání přechodů (možných tahů) ovlivní efektivitu prořezávání
- ➜ v případě "nejlepšího" uspořádání časová složitost$= O(b^{m/2})$
  - ⇒ zdvojí hloubku prohledávání
  - ⇒ může snadno dosáhnout hloubky 8 v šachu, což už je použitelná úroveň

označení $\alpha - \beta$:

- ➜ $\alpha \ldots$ doposud nejlepší hodnota pro MAXe
- ➜ $\beta \ldots$ doposud nejlepší hodnota pro MINa
- ➜ $<\alpha, \beta> \ldots$ interval ohodnocovací funkce v průběhu výpočtu (na začátku $<-\infty, \infty>$)
- ➜

minimax $\ldots V(P)$	$\alpha - \beta \ldots V(P, \alpha, \beta)$
když $V(P) \leq \alpha$	$V(P, \alpha, \beta) = \alpha$
když $\alpha < V(P) < \beta$	$V(P, \alpha, \beta) = V(P)$
když $V(P) \geq \beta$	$V(P, \alpha, \beta) = \beta$

Algoritmus Alfa-Beta prořezávání

```
alphabeta( Pos, Alpha, Beta, GoodPos, Val) :- moves( Pos, PosList), !,
    boundedbest( PosList, Alpha, Beta, GoodPos, Val);
    staticval ( Pos, Val).                          % Static value of Pos

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)  :- !.    % No other candidate
goodenough( _, Alpha, Beta, Pos, Val, Pos, Val)  :-
    min_to_move( Pos), Val > Beta, !                % Maximizer attained upper bound
    ; max_to_move( Pos), Val < Alpha, !.            % Minimizer attained lower bound
goodenough( PosList, Alpha, Beta, Pos, Val, GoodPos, GoodVal) :-
    newbounds( Alpha, Beta, Pos, Val, NewAlpha, NewBeta), % Refine bounds
    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, !.              % Maximizer increased lower bound
newbounds( Alpha, Beta, Pos, Val, Alpha, Val)  :-
    max_to_move( Pos), Val < Beta, !.               % Minimizer decreased upper bound
newbounds( Alpha, Beta, _, _, Alpha, Beta).         % Otherwise bounds unchanged

betterof( Pos, Val, Pos1, Val1, Pos, Val)  :- min_to_move( Pos), Val > Val1, !
    ; max_to_move( Pos), Val < Val1, !.             % Pos better than Pos1
betterof( _, _, Pos1, Val1, Pos1, Val1).            % Otherwise Pos1 better
```


Časové omezení

předpokládejme, že máme 100 sekund + prozkoumáme 10^4 uzlů/s ⇒ 10^6 uzlů na 1 tah

řešení:

- ❑ **ohodnocovací funkce** odhad přínosu pozice
- ❑ **ořezávací test** (*cutoff test*) – např. hloubka nebo hodnota ohodnocovací funkce

Možnosti vylepšení Minimaxu

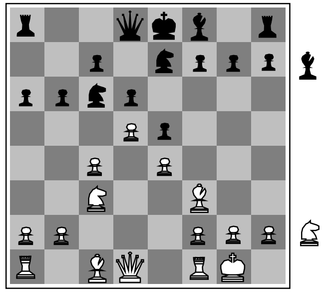
minimax_cutoff je stejný jako **minimax** kromě:

1. koncový test ⟶ *ořezávací test*
2. utilitární funkce ⟶ *ohodnocovací funkce*

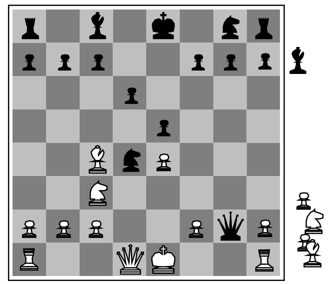
další možnosti vylepšení:

- ➜ vyhodnocovat pouze klidné stavy (quiescent search)
- ➜ při vyhodnocování počítat s efektem horizontu – zvraty mimo prohledanou oblast
- ➜ dopředné ořezávání – některé stavy se ihned zahazují
  bezpečné např. pro symetrické tahy nebo pro tahy hluboko ve stromu

Ohodnocovací funkce



Černý na tahu
Bílý ma o něco lepší pozici



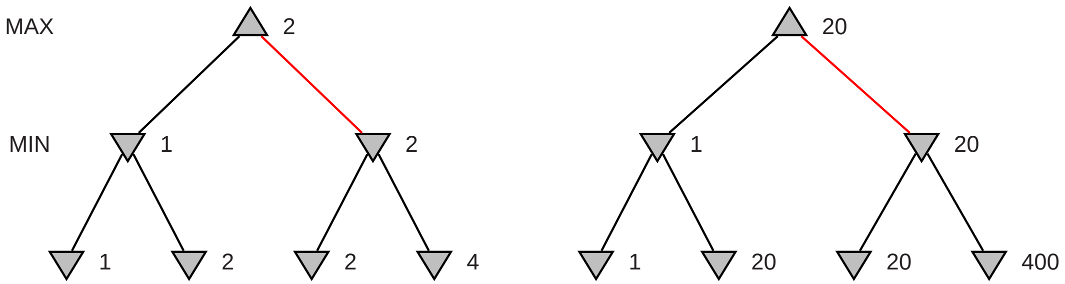
Bílý na tahu
Černý vítězí

Pro šachy typicky lineární vážený součet rysů

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

např. $w_1 = 9$
$f_1(s) = (\text{počet bílých královen}) - (\text{počet černých královen})$
$\ldots$

Ohodnocovací funkce – odchylky



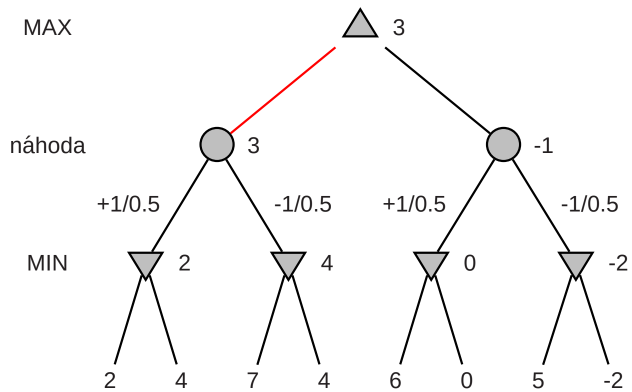
chová se stejně pro libovolnou monotónní transformaci fuknce $Eval$

záleží pouze na uspořádání ⟶ ohodnocení v deterministické hře funguje jako ordinální funkce

Nedeterministické hry

náhoda ⟵ hod kostkou, hod mincí, míchání karet

příklad – 1 tah s házení mincí:



Algoritmus Minimax pro nedeterministické hry

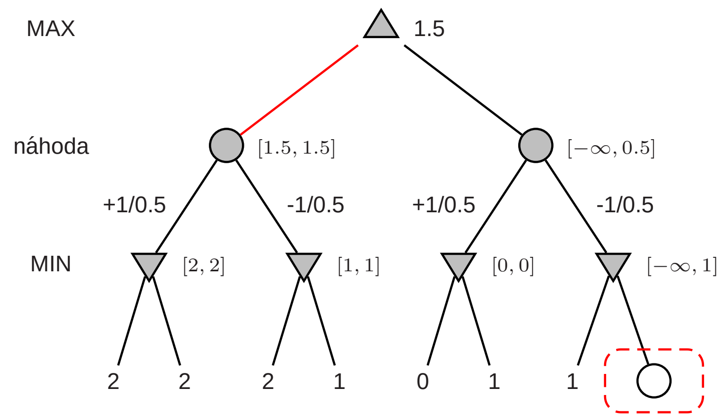
expect_minimax … počítá perfektní hru s přihlédnutím k náhodě

rozdíl je pouze v započítání uzlů *náhoda*:

$$\text{expect\_minimax}(n) = \begin{cases} \text{utility}(n) & \text{pro koncový stav } n \\ \max_{s \in \text{moves}(n)} \text{expect\_minimax}(s) & \text{pro MAX uzel } n \\ \min_{s \in \text{moves}(n)} \text{expect\_minimax}(s) & \text{pro MIN uzel } n \\ \sum_{s \in \text{moves}(n)} P(s) \cdot \text{expect\_minimax}(s) & \text{pro uzel náhody } n \end{cases}$$

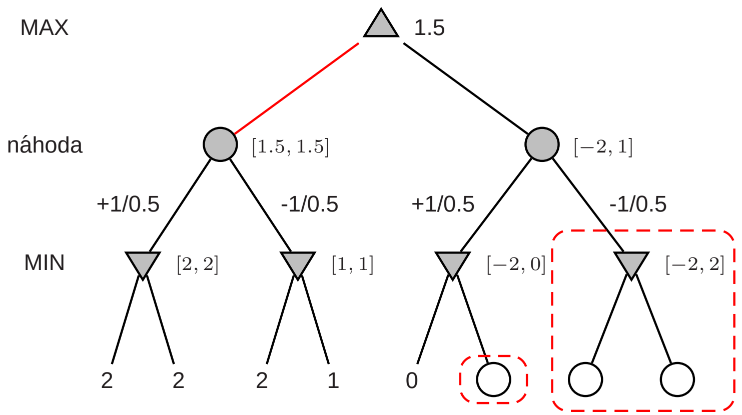
Prořezávání v nedeterministických hrách

je možné použít upravené Alfa-Beta prořezávání



Prořezávání v nedeterministických hrách pokrač.

pokud je možno dopředu stanovit limity na ohodnocení listů $\longrightarrow$ ořezávání je větší



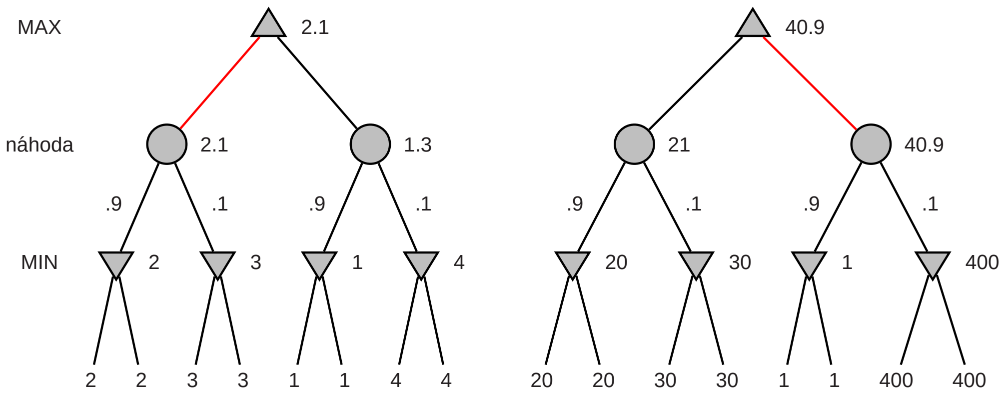
Nedeterministické hry v praxi

- hody kostkou zvyšují b $\longrightarrow$ se dvěma kostkami 21 možných výsledků
- backgammon – 20 legálních tahů:

$$\text{hloubka } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

- jak se zvyšuje hloubka $\longrightarrow$ pravděpodobnost dosažení zvoleného uzlu klesá
  $\Rightarrow$ význam prohledávání se snižuje
- alfa-beta prořezávání je mnohem méně efektivní
- program *TDGammon* používá prohledávání do hloubky 2 + velice dobrou $Eval$ funkci
  $\approx$ dosahuje úrovně světového šampionátu

Odchylka v ohodnocení nedeterministických her



chování je zachováno pouze pro pozitivní lineární transformaci fuknce $Eval$

$Eval$ u nedeterministických her by tedy měla proporcionálně odpovídat očekávanému výnosu

Hry s nepřesnými znalostmi

- např. karetní hry $\longrightarrow$ neznáme počáteční namíchání karet oponenta
- obvykle můžeme spočítat pravděpodobnost každého možného rozdání
- zjednodušeně – jako jeden velký hod kostkou na začátku
- prohledáváme ovšem ne *reálný stavový prostor*, ale domnělý stavový prostor
- program *GIB* vyhrál šampionát v roce 2000:
  1. generuje 100 rozdání karet konzistentních s daným podáním
  2. vybírá akci, která je v průměru nejlepší

Logický agent, výroková logika

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

➜ Logický agent
➜ Wumpusova jeskyně
➜ Logika
➜ Výroková logika
➜ Důkazové metody

Logický agent

logický agent = agent využívající znalosti (*knowledge-based agent*)

2 koncepty:
- – reprezentace znalostí (*knowledge representation*)
- – vyvozování znalostí (*knowledge reasoning*) ⟶ inference

rozdíly od prohledávání stavového prostoru:

➜ znalost při prohledávání stavového prostoru ⟶ jen zadané funkce (přechodová funkce, cílový test,...)

➜ znalosti logického agenta ⟶ obecná forma umožňující kombinace těchto znalostí

obecné znalosti – důležité v částečně pozorovatelných prostředích (*partially observable* environments)

flexibilita logického agenta:
➜ schopnost řešit i nové úkoly
➜ možnost učení nových znalostí
➜ úprava stávajících znalostí podle stavu prostředí

Báze znalostí

komponenty logického agenta:

inferenční stroj (inference engine)	⟵ algoritmy nezávislé na doméně
báze znalostí (knowledge base)	⟵ "informace" o doméně

báze znalostí (KB) = množina vět (*tvrzení*) vyjádřených v jazyce reprezentace znalostí

obsah báze znalostí:

➜ na začátku – tzv. znalosti pozadí (*background knowledge*)

➜ průběžně doplňované znalosti ⟶ úkol **tell(+KB,+Sentence)**

akce logického agenta:

```
% kb_agent_action (+KB,+ATime,+Percept,−Action,−NewATime)
kb_agent_action(KB,ATime,Percept,Action,NewATime):−
    make_percept_sentence(Percept,ATime,Sentence),
    tell (KB,Sentence),                          % přidáme výsledky pozorování do KB
    make_action_query(ATime,Query),
    ask(KB,Query,Action),                        % zeptáme se na další postup
    make_action_sentence(Action,ATime,ASentence),
    tell (KB,ASentence),                         % přidáme informace o akci do KB
    NewATime is ATime + 1.
```


Návrh logického agenta

přístupy k tvorbě agenta (systému) – deklarativní × procedurální (kombinace obou)

návrh agenta ⟶ víc pohledů:

❑ **znalostní hledisko** – tvorba agenta ⟶ zadání znalostí pozadí, znalostí domény a cílového požadavku

například automatické taxi

- znalost mapy, dopravních pravidel, ...
- požadavek – dopravit zákazníka na FI MU Brno

❑ **implementační hledisko** – jaké datové struktury KB obsahuje + algoritmy, které s nimi manipulují

agent musí umět:
➜ reprezentovat stavy, akce, ...
➜ zpracovat nové vstupy z prostředí
➜ aktualizovat svůj vnitřní popis světa
➜ odvodit skryté informace o stavu světa
➜ odvodit vlastní odpovídající akce

Popis světa – PEAS

zadání světa rozumného agenta:

- **míra výkonnosti** (*Performance measure*)
  plus body za dosažené (mezi)cíle, pokuty za nežádoucí následky
- **prostředí** (*Environment*)
  objekty ve světě, se kterými agent musí počítat, a jejich vlastnosti
- **akční prvky** (*Actuators*)
  možné součásti činnosti agenta, jeho akce se skládají z použití těchto prvků
- **senzory** (*Sensors*)
  zpětné vazby akcí agenta, podle jejich výstupů se tvoří další akce

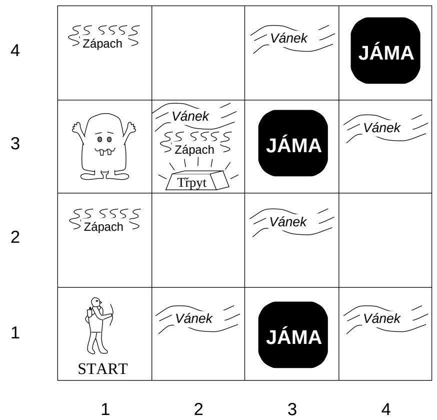
např. zmiňované automatické taxi:

míra výkonnosti	doprava na místo, vzdálenost, bezpečnost, bez přestupků, komfort, ...
prostředí	ulice, křižovatky, účastníci provozu, chodci, počasí, ...
akční prvky	řízení, plyn, brzda, houkačka, blinkry, komunikátory, ...
senzory	kamera, tachometr, počítač kilometrů, senzory motoru, GPS, ...

Wumpusova jeskyně

PEAS zadání Wumpusovy jeskyně:

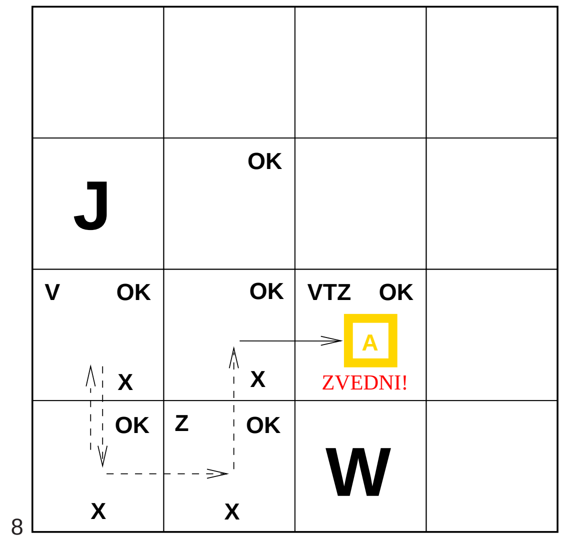
- **P – míra výkonnosti**
  zlato +1000, smrt -1000, -1 za krok, -10 za užití šípu
- **E – prostředí**
  Místnosti vedle Wumpuse zapáchají
  V místnosti vedle jámy je vánek
  V místnosti je zlato ⇔ je v ní třpyt
  Výstřel zabije Wumpuse, pokud jsi obrácený k němu
  Výstřel vyčerpá jediný šíp, který máš
  Zvednutím vezmeš zlato ve stejné místnosti
  Položení odloží zlato v aktuální místnosti
- **A – akční prvky**
  Otočení vlevo, Otočení vpravo, Krok dopředu,
  Zvednutí, Položení, Výstřel
- **S – senzory**
  Vánek, Třpyt, Zápach, Náraz do zdi, Chroptění Wumpuse



Vlastnosti problému Wumpusovy jeskyně

pozorovatelné	ne, jen lokální vnímání
deterministické	ano, přesně dané výsledky
episodické	ne, sekvenční na úrovni akcí
statické	ano, Wumpus a jámy se nehýbou
diskrétní	ano
více agentů	ne, Wumpus je spíše vlastnost prostředí

Průzkum Wumpusovy jeskyně



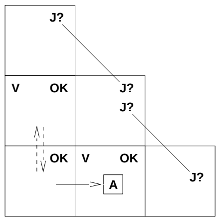
A	=	Agent
V	=	Vánek
T	=	Třpyt
OK	=	bezpečí
J	=	Jáma
Z	=	Zápach
X	=	navštíveno
W	=	Wumpus

Průzkum Wumpusovy jeskyně – problémy

základní vlastnost logického vyvozování:

Kdykoliv agent dospěje k závěru z daných informací $\longrightarrow$ tento závěr je zaručeně správný, pokud jsou správné dodané informace.

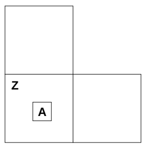
obtížné situace:



Vánek v $(1,2)$ i v $(2,1)$ $\Rightarrow$ žádná bezpečná akce

Při předpokladu uniformní distribuce děr

$\longrightarrow$ díra v $(2,2)$ má pravděpodobnost 0.86, na krajích 0.31



Zápach v $(1,1)$ $\Rightarrow$ nemůže se pohnout

je možné použít donucovací strategii (*strategy of coercion*):

1. Výstřel jedním ze směrů
2. byl tam Wumpus $\Rightarrow$ je mtrvý $\Rightarrow$ bezpečné
3. nebyl tam Wumpus $\Rightarrow$ bezpečný směr

Logika

Logika = *syntaxe* a *sémantika* formálního jazyka pro reprezentaci informací umožňující vyvozování závěrů

Syntaxe definuje všechny *dobře utvořené věty* jazyka

Sémantika definuje "*význam*" vět $\Rightarrow$ definuje pravdivost vět v jazyce (v závislosti na *možném světě*)

např. jazyk aritmetiky:

- $x+2 \geq y$ je dobře utvořená věta; $\quad x2+y>$ není věta
- $x+2 \geq y$ je pravda $\quad\Leftrightarrow\quad$ číslo $x+2$ není menší než číslo y
- $x+2 \geq y$ je pravda ve světě, kde $x=7,\ y=1$
- $x+2 \geq y$ je nepravda ve světě, kde $x=0,\ y=6$

zápis na papíře v libovolné syntaxi $\longrightarrow$ v KB se jedná o konfiguraci (částí) agenta

vlastní vyvozování $\longrightarrow$ generování a manipulace s těmito konfiguracemi

Důsledek

Důsledek (vyplývání, *entailment*) – jedna věc logicky vyplývá z druhé (je jejím důsledkem):

$$KB \models \alpha$$

Z báze znalostí KB vyplývá věta α $\quad\Leftrightarrow\quad$ α je pravdivá ve *všech světech*, kde je KB pravdivá

např.:

- KB obsahuje věty – "Češi vyhráli"

  – "Slováci vyhráli"

  z KB pak vyplývá – "Buď Češi vyhráli nebo Slováci vyhráli"
- z $x+y=4$ vyplývá $4=x+y$

Důsledek je vztah mezi větami (*syntaxe*), který je založený na *sémantice*.

Model

možný svět = model . . . formálně strukturovaný (abstraktní) svět, umožňuje vyhodnocení pravdivosti

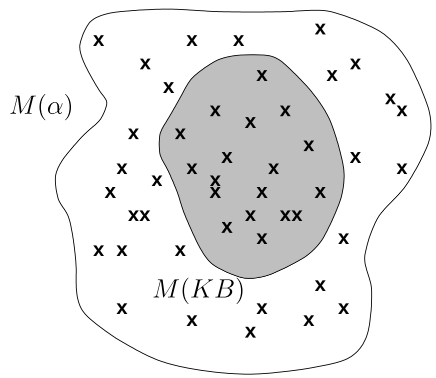
říkáme: $\quad m$ je model věty α $\quad\Leftrightarrow\quad$ α je pravdivá v m

$M(\alpha)$ . . . množina všech modelů věty α

$$KB \models \alpha \quad\Leftrightarrow\quad M(KB) \subseteq M(\alpha)$$

např.: $KB =$ "Češi vyhráli" $\wedge$ "Slováci vyhráli"

$\alpha =$ "Češi vyhráli"



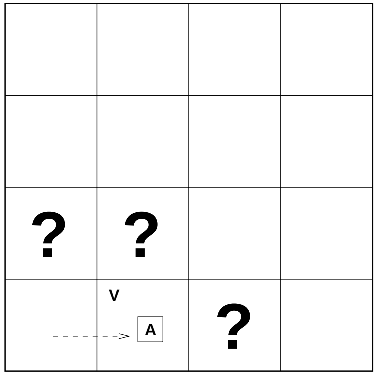
Vyplývání ve Wumpusově jeskyni

situace:

- v $[1,1]$ nedetekováno nic
- krok doprava, v $[2,1]$ Vánek

uvažujeme možné *modely* pro '**?**'

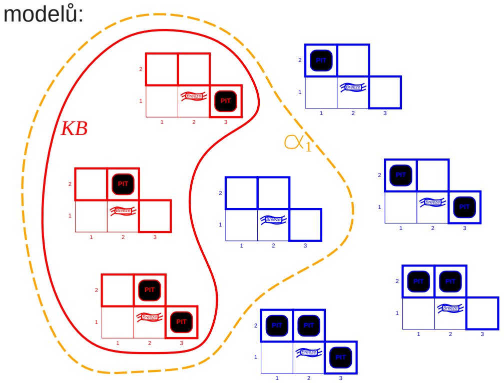
(budou nás zajímat jen Jámy)



3 pole s Booleovskými možnostmi $\{T, F\}$ $\Rightarrow$ $2^3 = 8$ možných modelů

Modely ve Wumpusově jeskyni

uvažujeme všech 8 možných modelů:



$KB =$ pravidla Wumpusovy jeskyně + pozorování

$\alpha_1 =$ "$[1,2]$ je bezpečné pole" $\quad KB \models \alpha_1$

$\alpha_2 =$ "$[2,2]$ je bezpečné pole" $\quad KB \not\models \alpha_2$

kontrola modelů $\longrightarrow$ jednoduchý způsob logické inference

Inference

Vyvozování požadovaných důsledků – inference

$KB \vdash_i \alpha \ldots$ věta α může být vyvozena z KB pomocí (procedury) i $\quad$ (i odvodí α z KB)

všechny možné důsledky KB jsou "kupka sena"; α je jehla

vyplývání = jehla v kupce sena; inference = její nalezení

Bezespornost: i je bezesporná $\Leftrightarrow$ $\forall KB \vdash_i \alpha \Rightarrow KB \models \alpha$

Úplnost: i je úplná $\Leftrightarrow$ $\forall KB \models \alpha \Rightarrow KB \vdash_i \alpha$

Vztah k *reálnému světu*:

> Pokud je KB pravdivá v reálném světě $\Rightarrow \forall$ věta α vyvozená z KB pomocí bezesporné inference je také pravdivá ve skutečném světě

Jestliže máme sémantiku "pravdivou" v reálném světě $\longrightarrow$ můžeme vyvozovat závěry o skutečném světě pomocí logiky

Výroková logika

Výroková logika – nejjednodušší logika, ilustruje základní myšlenky

- ❑ **výrokové symboly** $P_1, P_2, \ldots$ jsou věty
- ❑ **negace** – S je věta $\Rightarrow$ $\neg S$ je věta
- ❑ **konjunkce** – S_1 a S_2 jsou věty $\Rightarrow$ $S_1 \wedge S_2$ je věta
- ❑ **disjunkce** – S_1 a S_2 jsou věty $\Rightarrow$ $S_1 \vee S_2$ je věta
- ❑ **implikace** – S_1 a S_2 jsou věty $\Rightarrow$ $S_1 \Rightarrow S_2$ je věta
- ❑ **ekvivalence** – S_1 a S_2 jsou věty $\Rightarrow$ $S_1 \Leftrightarrow S_2$ je věta

Sémantika výrokové logiky

- každý model musí určit pravdivostní hodnoty výrokových symbolů
  např.: $m_1 = \{P_1 = false, P_2 = false, P_3 = true\}$
- pravidla pro vyhodnocení pravdivosti u složených výroků pro model m:

$\neg S$	je $true$	$\Leftrightarrow$	S	je $false$			
$S_1 \wedge S_2$	je $true$	$\Leftrightarrow$	S_1	je $true$	a	S_2	je $true$
$S_1 \vee S_2$	je $true$	$\Leftrightarrow$	S_1	je $true$	nebo	S_2	je $true$
$S_1 \Rightarrow S_2$	je $true$	$\Leftrightarrow$	S_1	je $false$	nebo	S_2	je $true$
tj.	je $false$	$\Leftrightarrow$	S_1	je $true$	a	S_2	je $false$
$S_1 \Leftrightarrow S_2$	je $true$	$\Leftrightarrow$	$S_1 \Rightarrow S_2$	je $true$	a	$S_2 \Rightarrow S_1$	je $true$

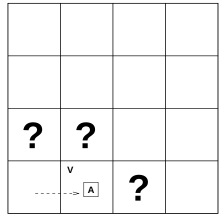
- rekurzivním procesem vyhodnotíme lib. větu:
  $\neg P_1 \wedge (P_2 \vee P_3) = true \wedge (false \vee true) = true \wedge true = true$

pravdivostní tabulka:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

Tvrzení pro Wumpusovu jeskyni

Definujeme výrokové symboly $J_{i,j}$ je pravda $\Leftrightarrow$ V $[i,j]$ je Jáma.
a $V_{i,j}$ je pravda $\Leftrightarrow$ V $[i,j]$ je Vánek.



báze znalostí KB:

- pravidlo pro $[1,1]$: R_1: $\neg J_{1,1}$
- pozorování: R_2: $\neg V_{1,1}$
  R_3: $V_{2,1}$
- pravidla pro vztah Jámy a Vánku:
  "Jámy způsobují Vánek ve vedlejších místnostech"

  R'_4: $V_{1,1} \Leftarrow (J_{1,2} \vee J_{2,1})$
  R'_5: $V_{2,1} \Leftarrow (J_{1,1} \vee J_{2,2} \vee J_{3,1})$

  "V poli je Vánek právě tehdy, když je ve vedlejším poli Jáma."

  R_4: $V_{1,1} \Leftrightarrow (J_{1,2} \vee J_{2,1})$
  R_5: $V_{2,1} \Leftrightarrow (J_{1,1} \vee J_{2,2} \vee J_{3,1})$
- $KB = R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$

Pravdivostní tabulka pro inferenci

$V_{1,1}$	$V_{2,1}$	$J_{1,1}$	$J_{1,2}$	$J_{2,1}$	$J_{2,2}$	$J_{3,1}$	KB	α_1
$false$	$false$	$false$	$false$	$false$	$false$	$false$	$false$	$true$
$false$	$false$	$false$	$false$	$false$	$false$	$true$	$false$	$true$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$false$	$true$	$false$	$false$	$false$	$false$	$false$	$false$	$true$
$false$	$true$	$false$	$false$	$false$	$false$	$true$	$\underline{true}$	$\underline{true}$
$false$	$true$	$false$	$false$	$false$	$true$	$false$	$\underline{true}$	$\underline{true}$
$false$	$true$	$false$	$false$	$false$	$true$	$true$	$\underline{true}$	$\underline{true}$
$false$	$true$	$false$	$false$	$true$	$false$	$false$	$false$	$true$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$true$	$true$	$true$	$true$	$true$	$true$	$true$	$false$	$false$

KB = pravidla Wumpusovy jeskyně + pozorování

α_1 = "$[1,2]$ je bezpečné pole"

Inference kontrolou modelů

Kontrola všech modelů *do hloubky* je bezesporná a úplná (pro konečný počet výrokových symbolů)

```
% tt_entails (+KB,+Alpha)
tt_entails (KB,Alpha):−proposition_symbols(Symbols,[KB,Alpha]),
    tt_check_all (KB,Alpha,Symbols,[]).

% tt_check_all (+KB,+Alpha,+Symbols,+Model)
tt_check_all (KB,Alpha,[],Model):−pl_true(KB,Model),!,pl_true(Alpha,Model).
tt_check_all (KB,Alpha,[],Model):−!,fail.
tt_check_all (KB,Alpha,[P|Symbols],Model):−
    tt_check_all (KB,Alpha,Symbols,[P−true|Model]),
    tt_check_all (KB,Alpha,Symbols,[P−false|Model]).
```

$O(2^n)$ pro n symbolů, NP-úplný problém

Logická ekvivalence

Dva výroky jsou logicky ekvivalentní právě tehdy, když jsou pravdivé ve stejných modelech:

$$\alpha \equiv \beta \quad \Leftrightarrow \quad \alpha \models \beta \text{ a } \beta \models \alpha$$

$(\alpha \wedge \beta)$	$\equiv$	$(\beta \wedge \alpha)$	komutativita $\wedge$
$(\alpha \vee \beta)$	$\equiv$	$(\beta \vee \alpha)$	komutativita $\vee$
$((\alpha \wedge \beta) \wedge \gamma)$	$\equiv$	$(\alpha \wedge (\beta \wedge \gamma))$	asociativita $\wedge$
$((\alpha \vee \beta) \vee \gamma)$	$\equiv$	$(\alpha \vee (\beta \vee \gamma))$	asociativita $\vee$
$\neg(\neg\alpha)$	$\equiv$	α	eliminace dvojí negace
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\beta \Rightarrow \neg\alpha)$	kontrapozice
$(\alpha \Rightarrow \beta)$	$\equiv$	$(\neg\alpha \vee \beta)$	eliminace implikace
$(\alpha \Leftrightarrow \beta)$	$\equiv$	$((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	eliminace ekvivalence
$\neg(\alpha \wedge \beta)$	$\equiv$	$(\neg\alpha \vee \neg\beta)$	de Morgan
$\neg(\alpha \vee \beta)$	$\equiv$	$(\neg\alpha \wedge \neg\beta)$	de Morgan
$(\alpha \wedge (\beta \vee \gamma))$	$\equiv$	$((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivita $\wedge$ nad $\vee$
$(\alpha \vee (\beta \wedge \gamma))$	$\equiv$	$((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivita $\vee$ nad $\wedge$

Platnost a splnitelnost

➜ Výrok je platný $\Leftrightarrow$ je pravdivý ve všech modelech

např.: $true, \quad A \vee \neg A, \quad A \Rightarrow A, \quad (A \wedge (A \Rightarrow B)) \Rightarrow B$

Platnost je spojena s inferencí pomocí věty o dedukci:

$KB \models \alpha \quad \Leftrightarrow \quad (KB \Rightarrow \alpha)$ je platný výrok

➜ Výrok je splnitelný $\Leftrightarrow$ je pravdivý v některých modelech

např.: $A \vee B, \quad C$

Výrok je nesplnitelný $\Leftrightarrow$ je nepravdivý ve všech modelech

např.: $A \wedge \neg A$

Splnitelnost je spojena s inferencí pomocí důkazu α sporem (*reductio ad absurdum*):

$KB \models \alpha \quad \Leftrightarrow \quad (KB \wedge \neg\alpha)$ je nesplnitelný

Důkazové metody

- **kontrola modelů**
  - procházení pravdivostní tabulky (vždycky exponenciální v n)
  - vylepšené prohledávání s navracením (*improved backtracking*), např. Davis–Putnam–Logemann–Loveland
  - heuristické prohledávání prostoru modelů (bezesporné, ale neúplné)
- **aplikace inferenčních pravidel**
  - legitimní (bezesporné) generování nových výroků ze starých
  - důkaz = sekvence aplikací inferenčních pravidel

    je možné použít inferenční pravidla jako operátory ve standardních prohledávacích algoritmech
  - typicky vyžaduje překlad vět do normální formy

Dopředné a zpětné řetězení

Hornovy klauzule: KB = konjunkce Hornových klauzulí

$$\text{Hornova klauzule} = \begin{cases} \text{výrokový symbol; nebo} \\ (\text{konjunkce symbolů}) \Rightarrow \text{symbol} \end{cases}$$

např.: $KB = C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$

pravidlo Modus Ponens – pro KB z Hornových klauzulí je úplné

$$\frac{\alpha_1, \ldots, \alpha_n, \qquad \alpha_1 \wedge \cdots \wedge \alpha_n \Rightarrow \beta}{\beta}$$

pravidla pro logickou ekvivalenci se taky dají použít pro inferenci

inference Hornových klauzulí $\longrightarrow$ algoritmus dopředného nebo zpětného řetězení

oba tyto algoritmy jsou přirozené a mají lineární časovou složitost

Dopředné řetězení

Idea: aplikuj pravidlo, jehož premisy jsou splněné v KB

přidej jeho důsledek do KB

pokračuj do doby, než je nalezena odpověď

KB:

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

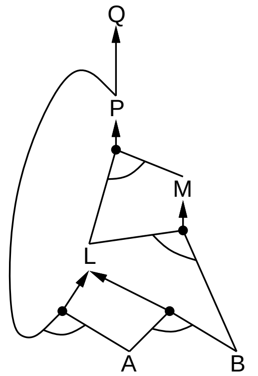
$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B

AND-OR graf KB:



Algoritmus dopředného řetězení

```
:- op( 800, fx, if ),
   op( 700, xfx, then),
   op( 300, xfy, or),
   op( 200, xfy, and).

forward :- new_derived_fact( P), !,                       % A new fact
   write( 'Derived: '), write( P), nl,
   assert( fact( P)),
   forward                                                 % Continue
   ;
   write( 'No more facts').                                % All facts derived

new_derived_fact( Concl) :- if Cond then Concl,            % A rule
   not(fact( Concl)),                                      % Rule's conclusion not yet a fact
   composed_fact( Cond).                                   % Condition true?

composed_fact( Cond) :- fact( Cond).                       % Simple fact
composed_fact( Cond1 and Cond2) :- composed_fact( Cond1),
   composed_fact( Cond2).                                  % Both conjuncts true
composed_fact( Cond1 or Cond2) :- composed_fact( Cond1)
   ; composed_fact( Cond2).
```


Dopředné řetězení – příklad

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

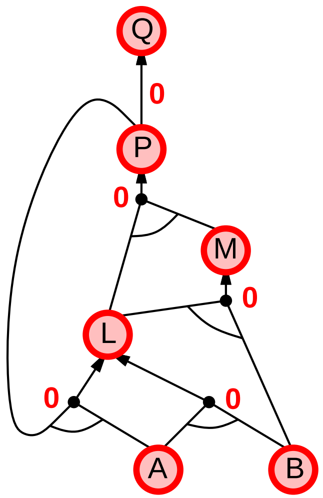
$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A

B



Zpětné řetězení

Idea: pracuje zpětně od dotazu q

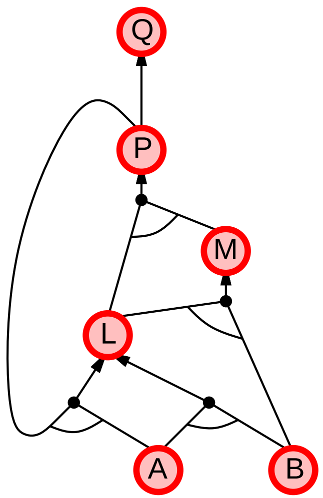
zkontroluj, jestli není q už známo

dokaž zpětným řetězením všechny premisy nějakého pravidla, které má q jako důsledek

kontrola cyklů – pro každý podcíl se nejprve podívej, jestli už nebyl řešen (tj. pamatuje si $true$ i $false$ výsledek)

Zpětné řetězení – příklad

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
A
B



Porovnání dopředného a zpětného řetězení

- **dopředné řetězení** je řízeno daty
  - automatické, nevědomé zpracování
  - např. rozpoznávání objektů, rutinní rozhodování
  - může udělat hodně nadbytečné práce bez vztahu k dotazu/cíli
- **zpětné řetězení** je řízeno dotazem
  - vhodné pro hledání odpovědí na konkrétní dotaz
  - např. “Kde jsou moje klíče?” “Jak se mám přihlásit na PGS?”
  - složitost zpětného řetězení může být mnohem menší než lineární vzhledem k velikosti KB

obecný inferenční algoritmus – rezoluce

zpracovává formule v konjunktivní normální formě (konjunkce disjunkcí literálů)

pro výrokovou logiku je rezoluce bezesporná a úplná

Logika prvního řádu a transparentní intenzionální logika (TIL)

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

➜ Predikátová logika prvního řádu
➜ Logická analýza přirozeného jazyka
➜ Transparentní intenzionální logika

Výhody a nevýhody výrokové logiky

☺ výroková logika je deklarativní: syntaxe přímo koresponduje s fakty

☺ výroková logika umožňuje zpracovávat částečné/disjunktivní/negované informace (což je víc, než umí většina datových struktur a databází)

☺ výroková logika je kompoziční:

význam $P_1 \wedge P_2$ je odvozen z významu P_1 a P_2

☺ ve výrokové logice je význam kontextově nezávislý (narozdíl od přirozeného jazyka, kde význam závisí na kontextu)

☹ výroková logika má velice omezenou expresivitu (narozdíl od přirozeného jazyka)

např. nemáme jak říct "Jámy způsobují Vánek ve vedlejších místnostech" jinak, než vyjmenovat odpovídající výrok pro každé pole

Predikátová logika prvního řádu

➜ *First-order predicate logic*, FOPL/PL1

➜ výroková logika $\longrightarrow$ svět obsahuje fakty $\times$ PL1 předpokládá, že svět obsahuje:

- **objekty** – lidi, domy, teorie, barvy, roky, ...
- **relace** – červený, kulatý, provčíselný, bratři, větší než, uvnitř, ...
- **funkce** – otec někoho, nejlepší přítel, plus jedna, začátek čeho, ...

➜ jiné možné logiky:

jazyk	ontologie	pravdivostní hodnoty
výroková logika	fakty	true/false/$\perp$
predikátová logika 1. řádu	fakty, objekty, relace	true/false/$\perp$
temporální logika	fakty, objekty, relace, čas	true/false/$\perp$
teorie pravděpodobnosti	fakty	míra pravděpodobnosti $\in [0,1]$
fuzzy logika	míra pravdivosti $\in [0,1]$	intervaly hodnot

Syntaxe predikátové logiky

➜ základní prvky –

konstanty	KingJohn, 2, RichardTheLionheart, ...
funktory predikátů	Brother, $>$, ...
funkce	Sqrt, LeftLegOf, ...
proměnné	$x,\ y,\ a,\ b,\ldots$
spojky	$\wedge\ \vee\ \neg\ \Rightarrow\ \Leftrightarrow$
rovnost	$=$
kvantifikátory	$\forall\ \exists$

➜ atomické formule –

predikáty	Brother(KingJohn, RichardTheLionheart)
složené termy	$>$ (Length(LeftLegOf(Richard)), Length(LeftLegOf(KingJohn)))

➜ složené formule – tvoří se z atomických formulí pomocí spojek

$$\neg S,\quad S_1 \wedge S_2,\quad S_1 \vee S_2,\quad S_1 \Rightarrow S_2,\quad S_1 \Leftrightarrow S_2$$

např. Sibling(KingJohn, Richard) $\Rightarrow$ Sibling(Richard, KingJohn)

$>(1,2) \vee \leq(1,2)$

$>(1,2) \wedge \neg{>}(1,2)$

Pravdivost v predikátové logice

pravdivost formule (sémantika) se určuje vzhledem k *modelu* a *interpretaci*

model obsahuje ≥ 1 objektů a relace mezi nimi

interpretace definuje vztah mezi syntaxí a modelem – určuje referenty pro:

konstantní symboly $\longrightarrow$ objekty

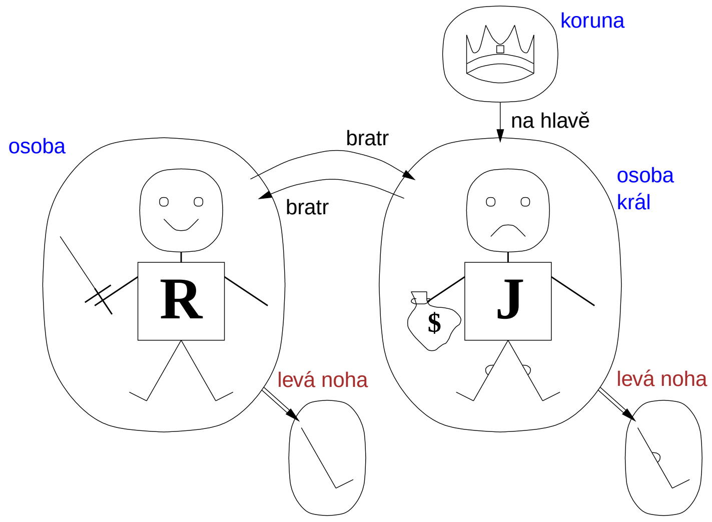
predikátové symboly $\longrightarrow$ relace

funkční symboly $\longrightarrow$ funkce

atomická formule **predikát(term$_1$, . . . , term$_n$)** je pravdivá $\Leftrightarrow$

$\Leftrightarrow$ *objekty* odkazované pomocí **term**$_1, \ldots,$ **term**$_n$ jsou v *relaci* pojmenované funktorem **predikát**.

Příklad modelu a interpretace ve FOPL



5 objektů, 2 binární relace, 3 unární relace (osoba, král, koruna) a 1 unární funkce (levá noha).

Inference ve FOPL

teoreticky můžeme určit všechny modely výčtem ze slovníku KB:

pro počet objektů $n = 1, \ldots, (\infty)$

pro každý k-ární predikát P_k ze slovníku

pro každou možnou k-ární relaci na n objektech

pro každý konstantní symbol C ze slovníku

pro každou volbu referenta pro C z n objektů . . .

prakticky je *kontrola modelů* nepoužitelná

inference je možná pouze podle inferenčních pravidel (dopředné/zpětné řetězení, rezoluce, . . .)

základní inferenční pravidlo – zobecněné Modus Ponens (*Generalized Modus Ponens, GMP*)

$$\frac{p_1{}',\ p_2{}',\ \ldots,\ p_n{}',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\textsc{Subst}(\theta, q)}$$

kde $\forall i\ \textsc{Subst}(\theta, p_i{}') = \textsc{Subst}(\theta, p_i)$
pro atomické formule $p_i, p_i{}'$ a q

– používá navíc unifikaci
– vzniká z MP pomocí liftingu
– využívá upravené verze inferenčních algoritmů – dopředné/zpětné řetězení, rezoluce

Univerzální kvantifikace

$\forall$⟨*proměnné*⟩ ⟨*formule*⟩

"Každý na FI MU je inteligentní:" $\forall x$ Na$(x,$ FI MU$) \Rightarrow$ inteligentní(x)

$\forall x\ P$ je pravdivé v modelu m $\Leftrightarrow$ P je pravdivá pro x = každý možný objekt z modelu m

zhruba odpovídá konjunkci instanciací P

Na(Petr, FI MU) $\Rightarrow$ inteligentní(Petr)
$\wedge$ Na(Honza, FI MU) $\Rightarrow$ inteligentní(Honza)
$\wedge$ Na(FI MU, FI MU) $\Rightarrow$ inteligentní(FI MU)
$\wedge$ . . .

Existenční kvantifikace

$\exists\langle$*proměnné*$\rangle$ $\langle$*formule*$\rangle$

"Někdo na MFF UK je inteligentní:" $\exists x$ Na$(x,$ MFF UK$)\wedge$ inteligentní(x)

$\exists x\ P$ je pravdivé v modelu m $\Leftrightarrow$ P je pravdivá pro x = nějaký objekt z modelu m

zhruba odpovídá disjunkci instanciací P

Na(Petr, MFF UK) $\wedge$ inteligentní(Petr)
$\vee$ Na(Honza, MFF UK) $\wedge$ inteligentní(Honza)
$\vee$ Na(MFF UK, MFF UK) $\wedge$ inteligentní(MFF UK)
$\vee$ ...

Vlastnosti kvantifikací

- pozor při použití kvantifikátorů na záměnu $\wedge$ a $\Rightarrow$:

	dobře	*špatně*	znamenalo by
"každý P je Q."	$\forall x\ P\Rightarrow Q$	$\forall x\ P\wedge Q$	"každý je P i Q."
"někdo P je Q."	$\exists x\ (P\wedge Q)$	$\exists x\ (P\Rightarrow Q)$	"někdo není P nebo je Q."

- $\forall x\forall y$ je stejné jako $\forall y\forall x$
  $\exists x\exists y$ je stejné jako $\exists y\exists x$
  $\exists x\forall y$ není stejné jako $\forall y\exists x$

  $\exists x\forall y$ má_rád(x,y) – "Existuje osoba, kterou má rád každý na světě."
  $\forall y\exists x$ má_rád(x,y) – "Každého na světě má alespoň jedna osoba ráda."

- dualita kvantifikátorů
  oba mohou být vyjádřeny pomocí druhého

  $\forall x$ má_rád$(x,$ zmrzlina$)$ $\quad\neg\exists x\ \neg$má_rád$(x,$ zmrzlina$)$
  $\exists x$ má_rád$(x,$ mrkev$)$ $\quad\neg\forall x\ \neg$má_rád$(x,$ mrkev$)$

Báze znalostí ve FOPL

předpokládejme, že agent ve Wumpusově jeskyni cítí Zápach a Vánek, ale nevidí Třpyt, nenarazil do zdi a nezabil Wumpuse v čase $t=5$:

TELL$(KB,$ Percept$([$Zápach, Vánek, nic, nic, nic$],5))$
ASK$(KB,\exists a$ Action$(a,5))$

tj. dotaz "Vyplývá nějaká akce z KB v čase $t=5$?"

odpověď: $true,\{a/$Výstřel$\}$ $\quad\longleftarrow$ substituce (hodnot proměnným)

pro větu S a substituci $\sigma\rightarrow S\sigma$ označuje výsledek aplikace σ na S:

$$\begin{aligned} S &= \text{chytřejší}(x,y)\\ \sigma &= \{x/\text{Petr}, y/\text{Honza}\}\\ S\sigma &= \text{chytřejší}(\text{Petr},\text{Honza})\end{aligned}$$

ASK(KB,S) vrací některá/všechna σ takové, že $KB\models S\sigma$

Báze znalostí pro Wumpusovu jeskyni

Vnímání:

$\forall v,tr,n,w,t$ Percept$([$Zápach$,v,tr,n,w],t)\Rightarrow$ Je_zápach(t)
$\forall z,v,n,w,t$ Percept$([z,v,$Třpyt$,n,w],t)\Rightarrow$ Máme_zlato(t)

Reflex:

$\forall t$ Máme_zlato$(t)\Rightarrow$ Action(Zvednutí$,t)$

Reflex s vnitřním stavem: neměli jsme už zlato?

$\forall t$ Máme_zlato$(t)\wedge\neg$Držím(Zlato$,t)\Rightarrow$ Action(Zvednutí$,t)$

Držím(Zlato$,t)$ není pozorovatelné $\Rightarrow$ je důležité držet si informace o vnitřních stavech

BÁZE ZNALOSTÍ PRO WUMPUSOVU JESKYNI pokrač.

Vyvozování skrytých skutečností:

- vlastnosti pozice:

  $\forall x, t$ Na_poli(Agent, x, t) $\wedge$ Je_zápach(t) $\Rightarrow$ Zapáchá(x)

  $\forall x, t$ Na_poli(Agent, x, t) $\wedge$ Je_vánek(t) $\Rightarrow$ S_vánkem(x)
- "V poli vedle Jámy je Vánek:"
  - diagnostické pravidlo – odvodí příčiny z následku

    $\forall y$ Je_vánek(y) $\Rightarrow \exists x$ Jáma(x) $\wedge$ Vedle(x, y)
  - příčinné pravidlo – odvodí výsledek z premisy

    $\forall x, y$ Jáma(x) $\wedge$ Vedle(x, y) $\Rightarrow$ Je_vánek(y)
  - ani jedno z nich není úplné

    např. příčinné pravidlo neříká, jestli v poli daleko od Jámy nemůže být Vánek
  - definice predikátu Je_vánek:

    $\forall y$ Je_vánek(y) $\Leftrightarrow$ [$\exists x$ Jáma(x) $\wedge$ Vedle(x, y)]

BÁZE ZNALOSTÍ PRO WUMPUSOVU JESKYNI – ROZHODOVÁNÍ

- počáteční podmínka v KB:

  Na_poli(Agent, $[1,1], S_0$)

  Na_poli(Zlato, $[1,2], S_0$)
- dotaz

  Ask($KB, \exists s$ Držím(Zlato, s))

  tj., "V jaké situaci budu držet Zlato?"
- situace jsou propojeny pomocí funkce $Result$:

  $Result(a, s)$ je situace, která je výsledkem činnosti a v s
- odpověď

  $\{s/Result($Zvednutí$, Result($Krok dopředu$, S_0))\}$

  tj., jdi dopředu a zvedni Zlato

SHRNUTÍ

logický agent aplikuje inferenci na bázi znalostí pro vyvození nových informací a tvorbu rozhodnutí

základní koncepty logiky:
- syntaxe: formální struktura vět
- sémantika: pravdivost vět podle modelů
- vyplývání: nutná pravdivost jedné věty v závislosti na druhé větě
- inference: vyvození věty z jiných vět
- bezespornost: když inference produkuje pouze vyplývající věty
- úplnost: když inference umí vyprodukovat všechny vyplývající věty

výroková logika nemá dostatečnou expresivitu

predikátová logika prvního řádu:
- objekty a relace jsou sémantická primitiva
- syntaxe: konstanty, funkce, predikáty, rovnost, kvantifikátory
- větší expresivita – dostatečná pro Wumpusovu jeskyni
- "poslední" logika, pro kterou existuje bezesporná a úplná inference (Gödelovy věty o neúplnosti)

LOGICKÁ ANALÝZA PŘIROZENÉHO JAZYKA

logická analýza PJ – analýza významu výrazů (vět) PJ

přirozený jazyk (čeština, angličtina, . . .) = nástroj pojmového uchopení reality

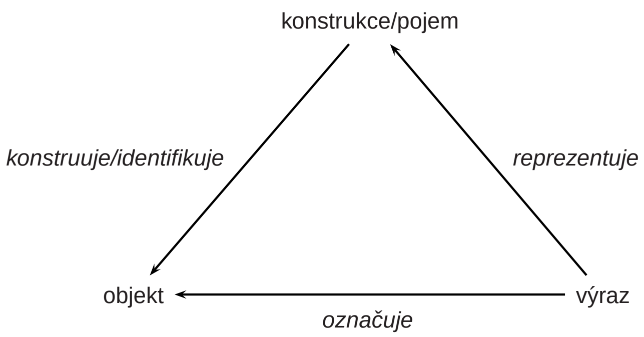
pojem – kritéria/procedury umožňující identifikovat různé konkrétní a abstraktní objekty (např. "planeta" – třída nebeských těles s určitými charakteristikami – obíhá po oběžné dráze kolem stálice, není zdrojem světla, . . .)

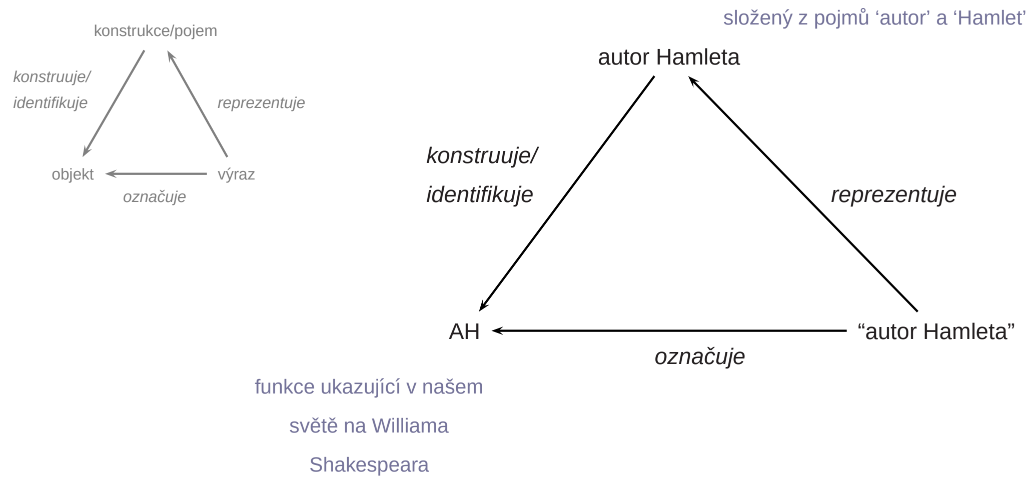
- pojem $\neq$ *výraz* – např. výrazy v různých jazycích často reprezentují stejný pojem

  (pojem("prvočíslo") $\equiv$ pojem("prime number"))
- pojem $\neq$ *představa* – představa je *subjektivní*, pojem je objektivní
- pojmy mohou identifikovat různé objekty:
  - jedno individuum – individuální pojmy (např. Petr, Pegas, prezident ČR)
  - třídu objektů – vlastnost (např. červený, šelma, hora)
  - n-člennou relaci – vztah (např. otec (někoho), křivdit (někdo někomu))
  - pravdivostní hodnotu – propozice (např. v Brně prší)
  - funkcionální přiřazení – empirické funkce (např. rychlost)
  - číslo – (fyzikální) veličiny (např. rychlost světla)

Vztah pojmu a výrazu

ve zjednodušené podobě: pojem odpovídá logické konstrukci





Omezenost predikátové logiky 1. řádu

dva omezující rysy:
- nedostatečná expresivita
- extenzionalismus

Expresivita: vyjadřovací síla jazyka

“Je-li barva stropu pokoje č. 3 uklidňující, je pokoj č. 3 vhodný pro pacienta X a není vhodný pro pacienta Y.”

analýza ve výrokové logice:

$$P \Rightarrow (Q \wedge \neg R)$$

P	“Barva stropu pokoje č. 3 je uklidňující.”
Q	“Pokoj č. 3 je vhodný pro pacienta X.”
R	“Pokoj č. 3 není vhodný pro pacienta Y.”

analýza v PL1:

$$U(B) \Rightarrow (V(P,X) \wedge \neg V(P,Y))$$

U	třída uklidňujících objektů
B	individuum ‘barva stropu pokoje č. 3’
V	relace mezi individuy ‘být vhodný pro’
P	individuum ‘pokoj č. 3’
X, Y	individua ‘pacient X’ a ‘pacient Y’

Nedostatečná expresivita PL1

Červená barva je krásnější než hnědá barva. *Kostka je červená.*

analýza v PL1:

$$Kr(\check{C}_1, H) \qquad \check{C}_2(Ko)$$

$\check{C}_1$	individuum ‘červená barva’
$\check{C}_2$	vlastnost individuí ‘být červený’ (třída červených objektů)

nelze vyjádřit $\check{C}_1 \equiv \check{C}_2$

EXTENZIONALISMUS PL1

Varšava *hlavní město Polska*

Varšava – jméno individua, jasně identifikovatelné a odlišitelné

hlavní město Polska – individuová role, momentálně identifikuje Varšavu, ale dříve to byl i Krakov

'hlavní město Polska'

- závisí na světě a čase
- pochopení významu, ale není vázané na znalost obsahu – tj. význam na světě a čase nezávisí

číslo X je větší než číslo Y *budova X je větší než budova Y*

matematické větší než – relace dvojic čísel, pevně daná

empirické větší než – vztah dvou individuí, který se může měnit v čase (otec a syn)

EXTENZIONALISMUS PL1 pokrač.

ano *V Brně prší*

ano – pravdivostní hodnota $true$

V Brně prší – propozice – označuje pravdivostní hodnotu, která se mění (alespoň) v čase

i když hodnota někdy závisí na světě a čase, samotný význam na nich nezávisí

EXTENZE A INTENZE

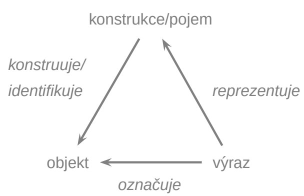
Definujeme:

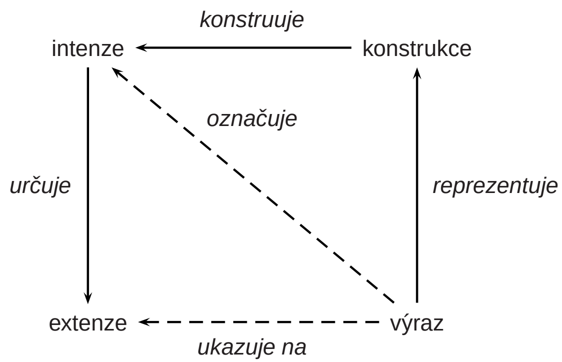
- **intenze** – objekty typu funkcí, jejichž hodnoty závisí na světě a čase
- **extenze** – ostatní objekty (na světě a čase nezávislé)

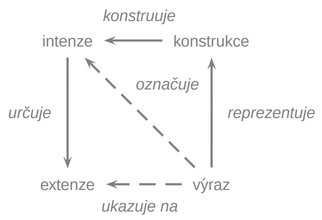
časté extenze a intenze:

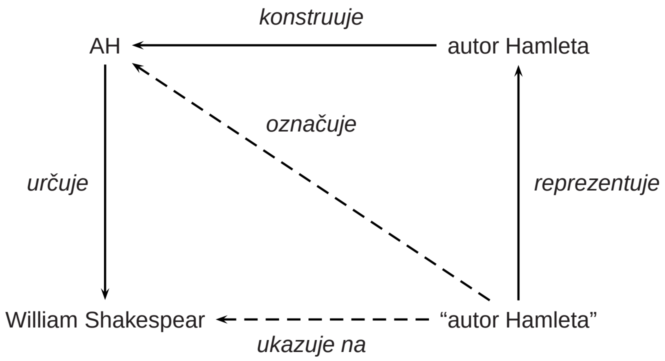
extenze	*intenze*
individua	individuové role
třídy	vlastnosti
relace	vztahy
pravdivostní hodnoty	propozice
funkce	empirické funkce
čísla	veličiny

ROZŠÍŘENÝ VZTAH VÝRAZU A VÝZNAMU U INTENZÍ









TRANSPARENTNÍ INTENZIONÁLNÍ LOGIKA

- *Transparent Intensional Logic*, TIL
- logický systém speciálně navržený pro zachycení významu výrazů PJ
- autor Pavel Tichý: *The Foundations of Frege's Logic*, de Gruyter, Berlin, New York, 1988.
- obdobná teorie – *Montagueho intenzionální logika* – Tichý ukazuje její nedostatky
- Tichý vychází z myšlenek – *Gottlob Frege* (1848 – 1925, logik) a *Alonzo Church* (1903 – 1995, teorie typů)
- vlastnosti:
  - rozvětvená typová hierarchie (s typy vyšších řádů)
  - temporální
  - intenzionální (intenze $\times$ extenze)
- transparentost:
  1. nositel významu (konstrukce) není prvek formálního aparátu, tento aparát pouze *studuje* konstrukce
  2. zachycení intenzionality je přesně popsáno z matematického hlediska

TYPY V TILU

typ objektu:

- základní typy – typová báze = $\{o, \iota, \tau, \omega\}$
- funcionální typy – funkce nad typovou bází

  např. ι, $((\iota\tau)\omega)$, $(o\iota)$, $(((o\iota)\tau)\omega)$, $((o\tau)\omega), \ldots$

  $((\alpha\tau)\omega) \ldots$ závislost na světě a čase, vyjadřuje intenze – zápis $\alpha_{\tau\omega}$
- typy vyšších řádů – obsahují i třídy konstrukcí řádu n – $*_n$

ZÁKLADNÍ TYPY TILU

umožňují přiřadit typ objektům z intenzionální báze jazyka – třída základních vlastností (barvy, rozměry, postoje, . . .) popisujících stav světa

- ❑ o (omikron, o) . . . pravdivostní hodnoty Pravda ($true$, T) a Nepravda ($false$, F)

  přesně odpovídají běžným logikám, typy logických operátorů – (oo), (ooo)
- ❑ ι (jota) . . . třída individuí

  individua ovšem ne jako kompletní objekty, ale jako numerická identifikace nestrukturované entity
- ❑ τ (tau) . . . třída časových okamžiků (jako časového kontinua)

  zachycení závislosti na čase; současně třída reálných čísel
- ❑ ω (omega) . . . třída možných světů

  zachycení empirické závislosti na stavu světa

Možné světy

termín možný svět – Gottfried Wilhelm von Leibniz (1646 – 1716, filozof a matematik)

požadavky na definici možného světa:

- soubor myslitelných faktů
- je konzistentní a maximální ze všech takových souborů
- je objektivní (nezávislý na individuálním názoru)

mezi možnými světy existuje právě jeden aktuální svět – jeho znalost ≡ vševědoucnost

možný svět v TILu = *rozhodovací systém*, pro ∀ prvek intenzionální báze obsahuje konzistentní přiřazení hodnot

příklad – realita s 2 objekty a 2 vlastnostmi (9 možných světů):

být hubený \ být tlustý	{Laurel, Hardy}	{Laurel}	{Hardy}	∅
{Laurel, Hardy}	×	×	×	w_1
{Laurel}	×	×	$\boxed{w_2}$	w_3
{Hardy}	×	w_4	×	w_5
∅	w_6	w_7	w_8	w_9

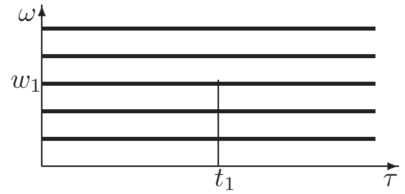
Princip intenzí v TILu

být hubený	… objekt typu $(o\iota)_{\tau\omega}$, funkce z možných světů a času do tříd individuí
w	… proměnná typu ω, možný svět
t	… proměnná typu τ, časový okamžik
[**být hubený** $w\ t$]	… konstruuje $(o\iota)$-objekt, třídu individuí, kteří mají ve světě w a čase t vlastnost být hubený (značíme **být hubený**$_{wt}$)

Americký prezident$_{w_{act}}$ (zkr. **P**$_{w_{act}}$) … ι_τ **P**$_{w_{act}t_0}$ … ι:

pokud aplikujeme jen w – získáme chronologii

intenzionální sestup – identifikace extenze pomocí intenze, světa w_1 a času t_1



Nejčastější typy

extenze			*intenze*		
individua	…	ι	individuové role	…	$\iota_{\tau\omega}$
třídy	…	$(o\iota)$	vlastnosti	…	$(o\iota)_{\tau\omega}$
relace	…	$(o\alpha\beta)$	vztahy	…	$(o\alpha\beta)_{\tau\omega}$
pravdivostní hodnoty	…	o	propozice	…	$o_{\tau\omega}$, π
funkce	…	$(\alpha\beta)$	empirické funkce	…	$(\alpha\beta)_{\tau\omega}$
čísla	…	τ	veličiny	…	$\tau_{\tau\omega}$

Konstrukce

konstrukce v TILu:

❑ **proměnná** typu α, v závislosti na valuaci konstruuje α-objekt

$x \dots \iota$

❑ **trivializace** objektu A typu α, konstruuje právě objekt A

^{0}A … α **A** … α

❑ **aplikace** konstrukce $X \dots (\alpha\beta_1 \dots \beta_n)$ na konstrukce $Y_1,\dots,Y_n$ typů $\beta_1, \dots, \beta_n$, konstruuje objekt typu α

$[XY_1 \dots Y_n] \dots \alpha$

❑ **abstrakce** konstrukce $Y \dots \alpha$ na proměnných $x_1,\dots,x_n$ typů $\beta_1, \dots, \beta_n$, konstruuje objekt/funkci typu $(\alpha\beta_1 \dots \beta_n)$

$\lambda x_1 \dots x_n[Y] \dots (\alpha\beta_1 \dots \beta_n)$

Příklady analýzy podstatných jmen

pes, člověk	$x \ldots \iota$: $\mathbf{pes}_{wt} x$, pes$/(o\iota)_{\tau\omega}$	individuum z dané třídy individuí
prezident	prezident$/\iota_{\tau\omega}$	individuová role
volitelnost	volitelnost$/(o\iota_{\tau\omega})_{\tau\omega}$	vlastnost individuové role
výška	výška$/(\tau\iota)_{\tau\omega}$	empirická funkce
výrok, tvrzení	$p \ldots *_n$: $\mathbf{výrok}_{wt} p$, výrok$/(o*_n)_{\tau\omega}$	konstrukce propozice z dané třídy konstrukcí propozic
válka, smích, zvonění	válka$/(o(o\pi))_{\omega}$	třída epizod – aktivita, která koresponduje se slovesem
leden, podzim	leden$/(o(o\tau))$	třída časových okamžiků — časové intervaly

Příklady přínosu TILu

- propoziční postoje

  Petr říká, že Tom věří, že Země je kulatá.

  $$\lambda w \lambda t \Big[\textbf{říká}_{wt} Petr^0 \Big[\lambda w \lambda t \big[\textbf{věří}_{wt} Tom^0 \big[\lambda w \lambda t [\textbf{kulatá}_{wt} \textbf{Země}] \big] \big] \Big] \Big]$$

- existence neexistujícího

  Pes existuje. *Jednorožec neexistuje.*

  v PL1: $\cancel{\exists x(x = \text{pes})}$ $\cancel{\neg\exists x(x = \text{jednorožec})}$

  $(\text{jednorožec} = \text{jednorožec}) \Rightarrow (\exists x(x = \text{jednorožec}))$

  v TILu:

  $$(*) \qquad \lambda w \lambda t \big[{}^0\neg [Ex_{wt}\, \textbf{jednorožec}]\big], \qquad Ex \stackrel{df}{=} \lambda w \lambda t \lambda p \Big[{}^0{\textstyle\sum}_\iota \big[\lambda x [p_{wt}\, x]\big]\Big]$$

  $$Ex \ldots (o(o\iota)_{\tau\omega})_{\tau\omega}$$

  $(*)$ … "třída všech individuí s vlastností 'být jednorožcem' je v daném světě a čase prázdná."

- intenzionalita, vlastnosti vlastností, analýza epizod, analýza gramatického času, …

Učení, rozhodovací stromy, neuronové sítě

Aleš Horák

E-mail: **hales@fi.muni.cz**

http://nlp.fi.muni.cz/uui/

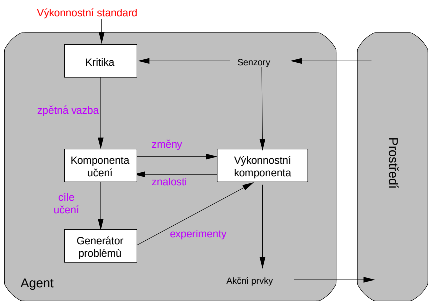
Obsah:

- ➜ Učení
- ➜ Rozhodovací stromy
- ➜ Neuronové sítě

Učení

- ➜ učení je klíčové pro neznámé prostředí (kde návrhář není vševědoucí)
- ➜ učení je také někdy vhodné jako metoda konstrukce systému – vystavit agenta realitě místo přepisování reality do pevných pravidel
- ➜ učení agenta – využití jeho vjemů z prostředí nejen pro vyvození další akce
- ➜ učení modifikuje rozhodovací systém agenta pro zlepšení jeho výkonnosti

Učící se agent



příklad automatického taxi:

- ❏ **Výkonnostní komponenta** – obsahuje znalosti a postupy pro výběr akcí pro vlastní řízení auta
- ❏ **Kritika** – sleduje reakce okolí na akce taxi. Např. při rychlém přejetí 3 podélných pruhů zaznamená a předá pohoršující reakce dalších řidičů
- ❏ **Komponenta učení** – z hlášení Kritiky vyvodí nové pravidlo, že takové přejíždění je nevhodné, a modifikuje odpovídajícím způsobem Výkonnostní komponentu
- ❏ **Generátor problémů** – zjišťuje, které oblasti by mohly potřebovat vylepšení a navrhuje experimenty, jako je třeba brždění na různých typech vozovky

Komponenta učení

návrh komponenty učení závisí na několika atributech:

- jaký typ výkonnostní komponenty je použit
- která funkční část výkonnostní komponenty má být učena
- jak je tato funkční část reprezentována
- jaká zpětná vazba je k dispozici

příklady:

výkonnostní komponenta	funkční část	reprezentace	zpětná vazba
Alfa-beta prohledávání	vyhodnocovací funkce	vážená lineární funkce	výhra/prohra
Logický agent	určení akce	axiomy $Result$	výsledné skóre
Reflexní agent	váhy preceptronu	neuronová síť	správná/špatná akce

učení s dohledem (*supervised learning*) $\times$ bez dohledu (*unsupervised learning*)

- ❏ **s dohledem** – učení funkce z příkladů vstupů a výstupů
- ❏ **bez dohledu** – učení vzorů na vstupu vzhledem k reakcím prostředí
- ❏ **posílené** (*reinforcement learning*) – nejobecnější, agent se učí podle odměn/pokut

Induktivní učení

známé taky jako věda ☺

nejjednodušší forma – učení funkce z příkladů (agent je tabula rasa)

f je cílová funkce

příklad je dvojice $x, f(x)$ např.

O	O	×
	×	
×		

$, \quad +1$

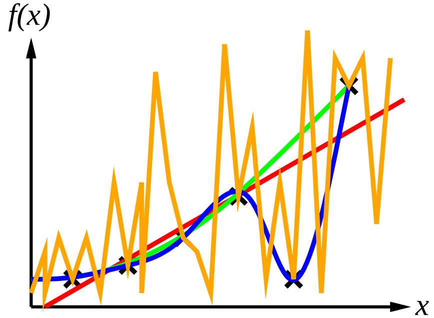
úkol indukce: najdi hypotézu h
takovou, že $h \approx f$
pomocí sady trénovacích příkladů

Metoda induktivního učení

zkonstruuj/uprav h, aby souhlasila s f na trénovacích příkladech

h je konzistentní $\Leftrightarrow$ souhlasí f f na všech příkladech

např. hledání křivky:



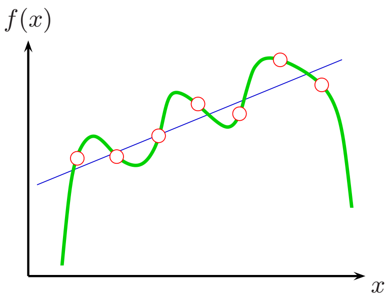
pravidlo Ockhamovy břitvy – maximalizovat kombinaci konzistence a jednoduchosti (*nejjednodušší ze správných je nejlepší*)

Metoda induktivního učení pokrač.

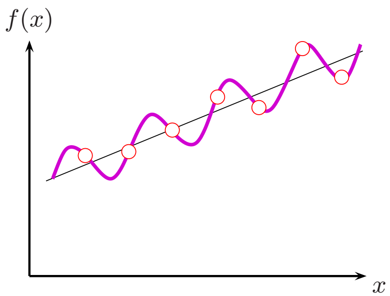
hodně záleží na prostoru hypotéz, jsou na něj protichůdné požadavky:
- pokrýt co největší množství hledaných funkcí
- udržet nízkou výpočetní složitost hypotézy

např.

a)



b)



- stejná sada 7 bodů
- nejmenší konzistentní polynom – polynom 6-tého stupně (7 parametrů)
- může být výhodnější použít nekonzistentní přibližnou lineární funkci
- přitom existuje konzistentní funkce $ax + by + c \sin x$

Atributová reprezentace příkladů

příklady popsané výčtem hodnot atributů (libovolných hodnot)

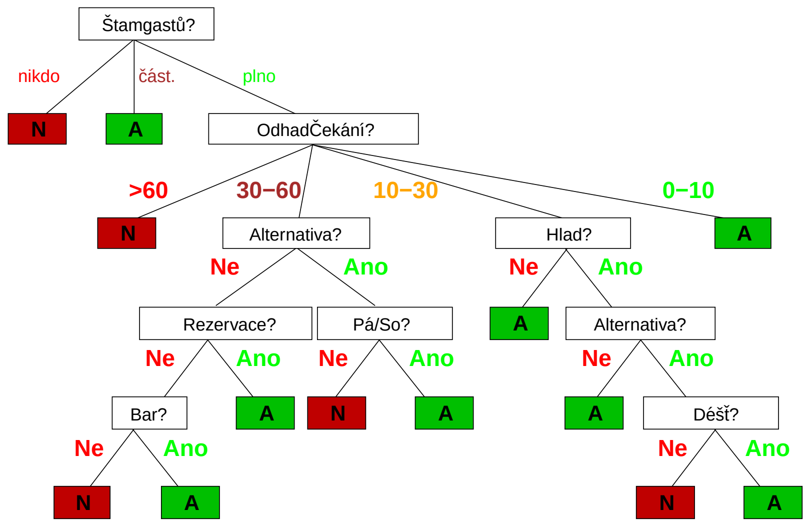
např. rozhodování, zda počkat na uvolnění stolu v restauraci:

Příklad	Atributy										počkat?
	Alt	Bar	$Pá/So$	$Hlad$	$Štam$	Cen	$Déšť$	Rez	Typ	$CekD$	
X_1	A	N	N	A	část.	$\$\$\$$	N	A	mexická	0–10	A
X_2	A	N	N	A	plno	$\$$	N	N	asijská	30–60	N
X_3	N	A	N	N	část.	$\$$	N	N	bufet	0–10	A
X_4	A	N	A	A	plno	$\$$	N	N	asijská	10–30	A
X_5	A	N	A	N	plno	$\$\$\$$	N	A	mexická	>60	N
X_6	N	A	N	A	část.	$\$\$$	A	A	pizzerie	0–10	A
X_7	N	A	N	N	nikdo	$\$$	A	N	bufet	0–10	N
X_8	N	N	N	A	část.	$\$\$$	A	A	asijská	0–10	A
X_9	N	A	A	N	plno	$\$$	A	N	bufet	>60	N
X_{10}	A	A	A	A	plno	$\$\$\$$	N	A	pizzerie	10–30	N
X_{11}	N	N	N	N	nikdo	$\$$	N	N	asijská	0–10	N
X_{12}	A	A	A	A	plno	$\$$	N	N	bufet	30–60	A

Ohodnocení tvoří klasifikaci příkladů – pozitivní (A) a negativní (N)

Rozhodovací stromy

jedna z možných reprezentací hypotéz – rozhodovací strom pro určení, jestli počkat na stůl:



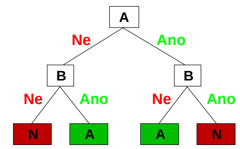
Vyjadřovací síla rozhodovacích stromů

rozhodovací stromy vyjádří libovolnou Booleovskou funkci vstupních atributů $\longrightarrow$ odpovídá výrokové logice

$\forall s$ počkat?$(s) \Leftrightarrow \big(P_1(s) \vee P_2(s) \vee \ldots \vee P_n(s)\big)$, kde $P_i(s) = \big(A_1(s) = V_1 \wedge \ldots \wedge A_m(s) = V_m\big)$

pro libovolnou Booleovskou funkci $\longrightarrow$ řádek v pravdivostní tabulce = cesta ve stromu (od kořene k listu)

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F



triviálně

pro libovolnou trénovací sadu existuje konzistentní rozhodovací strom s jednou cestou k listům pro každý příklad

ale takový strom pravděpodobně nebude generalizovat na nové příklady

chceme najít co možná kompaktní rozhodovací strom

Prostor hypotéz

1. vezměme pouze Booleovské atributy, bez dalšího omezení

   Kolik existuje různých rozhodovacích stromů s n Booleovskými atributy?

   = počet všech Booleovských funkcí nad těmito atributy

   = počet různých pravdivostních tabulek s 2^n řádky = 2^{2^n}

   např. pro 6 atributů existuje 18 446 744 073 709 551 616 různých rozhodovacích stromů

2. když se omezíme pouze na konjunktivní hypotézy ($Hlad \wedge \neg Déšť'$)

   Kolik existuje takových čistě konjunktivních hypotéz?

   každý atribut může být v pozitivní nebo negativní formě nebo nepoužit

   $\Rightarrow$ 3^n různých konjunktivních hypotéz (pro 6 atributů = 729)

prostor hypotéz s větší expresivitou

- zvyšuje šance, že najdeme přesné vyjádření cílové funkce
- ALE zvyšuje i počet možných hypotéz, které jsou konzistentní s trénovací množinou

  $\Rightarrow$ můžeme získat nižší kvalitu předpovědí (generalizace)

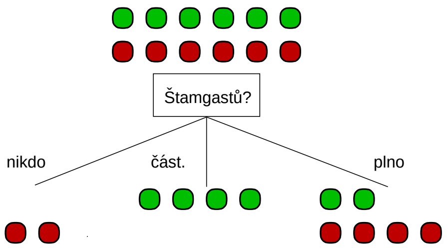
Učení ve formě rozhodovacích stromů

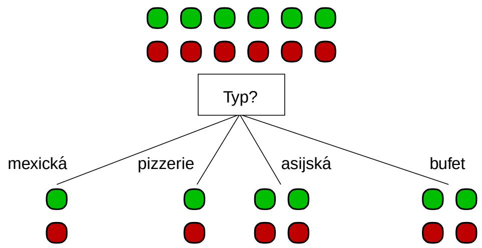
- **triviální konstrukce rozhodovacího stromu**
  - pro každý příklad v trénovací sadě přidej jednu cestu od kořene k listu
  - na stejných příkladech jako v trénovací sadě bude fungovat přesně
  - na nových příkladech se bude chovat náhodně – negeneralizuje vzory z příkladů, pouze kopíruje pozorování
- **heuristická konstrukce kompaktního stromu**
  - chceme najít nejmenší rozhodovací strom, který souhlasí s příklady
  - vlastní nalezení nejmenšího stromu je ovšem příliš složité

    $\longrightarrow$ heuristikou najdeme alespoň dostatečně malý ☺
  - hlavní myšlenka – vybíráme atributy pro test v co nejlepším pořadí

Výběr atributu

myšlenka – dobrý atribut rozdělí příklady na podmnožiny, které jsou (nejlépe) "všechny pozitivní" nebo "všechny negativní"





$Štamgastů?$ je lepší volba atributu $\longleftarrow$ dává lepší informaci o vlastní klasifikaci příkladů

Výběr atributu – míra informace

informace – odpovídá na otázku

čím méně dopředu vím o výsledku obsaženém v odpovědi $\longrightarrow$ tím více informace je v ní obsaženo

měřítko:

1 bit = odpověď na Booleovskou otázku s pravděpodobností odpovědi $\langle P(T)=\frac{1}{2}, P(F)=\frac{1}{2}\rangle$

pro pravděpodobnosti všech odpovědí $\langle P(v_1),\ldots,P(v_n)\rangle \longrightarrow$ míra informace v odpovědi obsažená

$$I(\langle P(v_1),\ldots,P(v_n)\rangle) = \sum_{i=1}^{n} -P(v_i)\log_2 P(v_i)$$

tato míra se také nazývá entropie

např. pro házení mincí: $I(\langle\frac{1}{2},\frac{1}{2}\rangle) = -\frac{1}{2}\log_2\frac{1}{2} - \frac{1}{2}\log_2\frac{1}{2} = 1$ bit

pro házení *falešnou* mincí, která dává na 99% vždy jednu stranu mince:

$$I(\langle\tfrac{1}{100},\tfrac{99}{100}\rangle) = -\tfrac{1}{100}\log_2\tfrac{1}{100} - \tfrac{99}{100}\log_2\tfrac{99}{100} = 0.08 \text{ bitů}$$

Použití míry informace pro výběr atributu

předpokládejme, že máme p pozitivních a n negativních příkladů

$\Rightarrow I\left(\langle\frac{p}{p+n},\frac{n}{p+n}\rangle\right)$ bitů je potřeba pro klasifikaci nového příkladu

např. pro $X_1,\ldots,X_{12}$ z volby čekání na stůl je $p=n=6$, takže potřebujeme 1 bit

výběr atributu – *kolik informace nám dá test na hodnotu atributu A?*

= rozdíl odhadu odpovědi před a po testu atributu

atribut A rozdělí sadu příkladů E na podmnožiny E_i (nejlépe, že $\forall$ potřebuje méně informace)

nechť E_i má p_i pozitivních a n_i negativních příkladů

$\Rightarrow$ je potřeba $I\left(\langle\frac{p_i}{p_i+n_i},\frac{n_i}{p_i+n_i}\rangle\right)$ bitů pro klasifikaci nového příkladu

$\Rightarrow$ očekávaný počet bitů přes $\forall$ větve je $Remainder(A) = \sum_i \frac{p_i+n_i}{p+n} I\left(\langle\frac{p_i}{p_i+n_i},\frac{n_i}{p_i+n_i}\rangle\right)$

$\Rightarrow$ výsledný zisk atributu A je $Gain(A) = I\left(\langle\frac{p}{p+n},\frac{n}{p+n}\rangle\right) - Remainder(A)$

výběr atributu = nalezení atributu s nejvyšší hodnotou $Gain(A)$

$Gain(Štamgastů?) \approx 0.541$ bitů $\qquad Gain(Typ?) = 0$ bitů

Algoritmus IDT – učení formou rozhodovacích stromů

```
% induce_tree( Attributes, Examples, Tree)
induce_tree( _, [], null) :- !.
induce_tree( _, [example( Class,_) | Examples], leaf( Class)) :-
    not (member( example( ClassX, _), Examples), ClassX \== Class), !. % ∀ příklady stejné klasifikace
induce_tree( Attributes, Examples, tree( Attribute, SubTrees)) :-
    choose_attribute( Attributes, Examples, Attribute), !,
    del( Attribute, Attributes, RestAtts),
    attribute( Attribute, Values),
    induce_trees( Attribute, Values, RestAtts, Examples, SubTrees).
induce_tree( _, Examples, leaf( ExClasses)) :-       % žádný užitečný atribut, list s stribucí klasifikací
    findall( Class, member( example( Class, _), Examples), ExClasses).

% induce_trees( Att, Values, RestAtts, Examples, SubTrees):
% najdi podstromy SubTrees pro podmnožiny příkladů Examples podle hodnot (Values) atributu Att
induce_trees( _, [], _, _, [] ).       % No attributes, no subtrees
induce_trees( Att, [Val1 | Vals], RestAtts, Exs, [Val1 : Tree1 | Trees]) :-
    attval_subset( Att = Val1, Exs, ExampleSubset),
    induce_tree( RestAtts, ExampleSubset, Tree1),
    induce_trees( Att, Vals, RestAtts, Exs, Trees).

% attval_subset( Attribute = Value, Examples, Subset):
% Subset je podmnožina příkladů z Examples, které splňují podmínku Attribute = Value
attval_subset( AttributeValue, Examples, ExampleSubset) :-
    findall( example( Class, Obj),
        (member( example( Class, Obj), Examples), satisfy( Obj, [ AttributeValue ])),
        ExampleSubset).
```

Algoritmus IDT – učení formou rozhodovacích stromů pokrač.

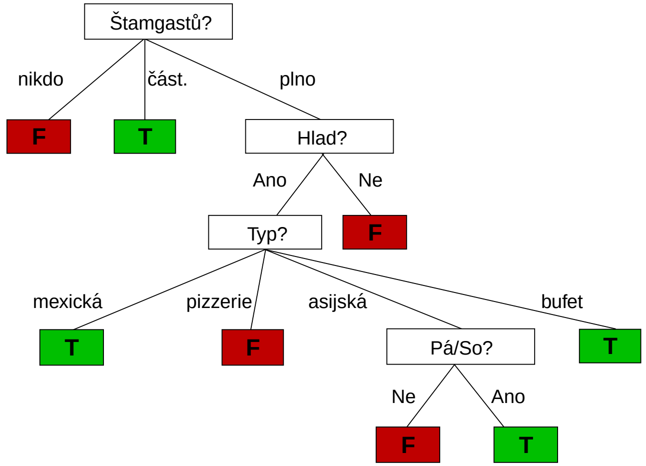
```
% satisfy( Object, Description)
satisfy ( Object, Conj)  :- not (member( Att = Val, Conj), member( Att = ValX, Object), ValX \== Val).

% vybíráme atribut podle "čistoty" množin, na které rozdělí příklady, setof je setřídí podle Impurity
choose_attribute( Atts, Examples, BestAtt) :-
    setof( Impurity/Att, (member( Att, Atts), impurity1(Examples, Att, Impurity)),  [MinImpurity/BestAtt|_]).

impurity1( Exs, Att, Imp) :- attribute( Att, AttVals), term_sum( Exs, Att, AttVals, 0, Imp).

% term_sum( Exs, Att, AttVals, PartialSum, Sum) - vážená suma "čistoty" přes ∀ hodnoty atributu Att
term_sum( _, _, [], Sum, Sum).
term_sum( Exs, Att, [Val | Vals], PartSum, Sum) :- length( Exs, N),
    findall( C, (member( example(C,Desc), Exs), satisfy( Desc, [Att=Val])), ExClasses),
    % ExClasses = seznam klasifikací (s opakováním) všech příkladů s Att=Val
    length( ExClasses, NV), NV > 0, !,
    findall( P, (bagof( 1, member( Class, ExClasses), L), length( L, NVC), P is NVC/NV), ClassDistribution),
    gini( ClassDistribution, GiniI),
    NewPartSum is PartSum + GiniI*NV/N,
    term_sum( Exs, Att, Vals, NewPartSum, Sum)
    ; term_sum( Exs, Att, Vals, PartSum, Sum).     % žádné příklady nesplňují Att = Val
```

% gini(ProbabilityList , GiniIndex) – míra "čistoty", $GiniIndex = \sum_{\forall i,j: i \neq j} P_i \cdot P_j \equiv 1 - \sum_{\forall i} P_i \cdot P_i$

```
gini( Probs, Index) :- square_sum( Probs, 0, SquareSum), Index is 1 - SquareSum.

square_sum( [], S, S).
square_sum( [P | Ps], PartS, S) :- NewPartS is PartS + P*P, square_sum( Ps, NewPartS, S).
```


IDT – výsledný rozhodovací strom

rozhodovací strom naučený z 12-ti příkladů:



podstatně jednodušší než strom "z tabulky příkladů"

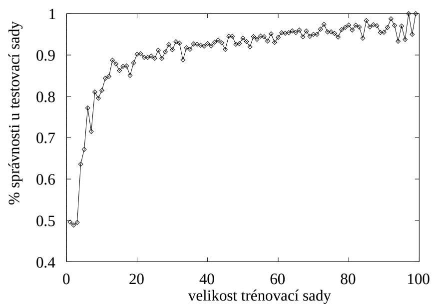
Hodnocení úspěšnosti učícího algoritmu

jak můžeme zjistit, zda $h \approx f$?
- dopředu – použít věty Teorie komputačního učení
- po naučení – kontrolou na jiné trénovací sadě

používaná metodologie:

1. vezmeme velkou množinu příkladů
2. rozdělíme ji na 2 množiny – trénovací a testovací
3. aplikujeme učící algoritmus na *trénovací* sadu, získáme hypotézu h
4. změříme procento příkladů v *testovací* sadě, které jsou správně klasifikované hypotézou h
5. opakujeme kroky 2–4 pro různé velikosti trénovacích sad a pro náhodně vybrané trénovací sady

křivka učení – závislost velikosti trénovací sady na úspěšnosti



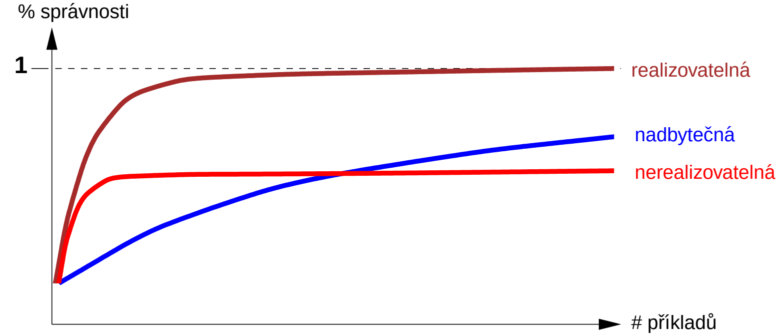
Hodnocení úspěšnosti učícího algoritmu pokrač.

tvar křivky učení závisí na ➜ je hledaná funkce realizovatelná × nerealizovatelná

funkce může být nerealizovatelná kvůli
- chybějícím atributům
- omezenému prostoru hypotéz

➜ naopak nadbytečné expresivitě

např. množství nerelevantních atributů



Induktivní učení – shrnutí

- ➜ učení je potřebné pro neznámé prostředí (a líné analytiky ☺)
- ➜ učící se agent – výkonnostní komponenta a komponenta učení
- ➜ metoda učení závisí na *typu výkonnostní komponenty*, dostupné *zpětné vazbě*, *typu* a *reprezentaci* části, která se má učením zlepšit
- ➜ u učení s dohledem – cíl je najít nejjednodušší hypotézu přibližně konzistentní s trénovacími příklady
- ➜ učení formou rozhodovacích stromů používá míru informace
- ➜ kvalita učení – přesnost odhadu změřená na testovací sadě

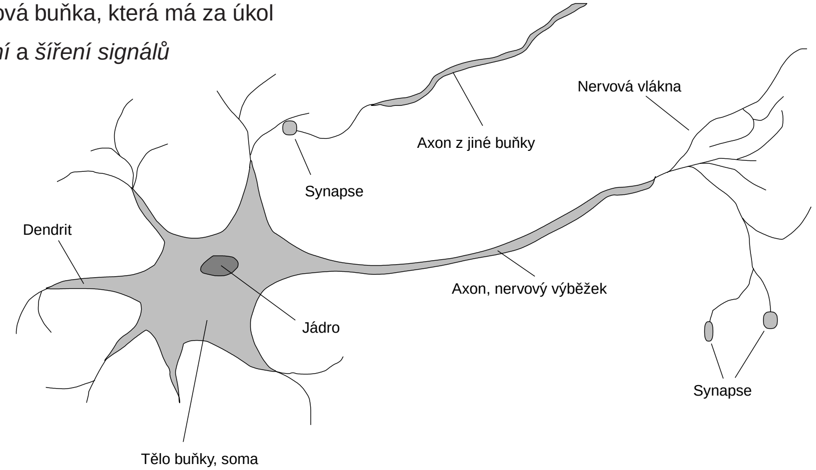
Neuron

mozek – 10^{11} neuronů > 20 typů, 10^{14} synapsí, 1ms–10ms cyklus

nosiče informace – signály = “výkyvy” elektrických potenciálů (se šumem)

neuron – mozková buňka, která má za úkol

sběr, *zpracování* a *šíření signálů*



Počítačový model – neuronové sítě

1943 – McCulloch & Pitts – matematický model neuronu

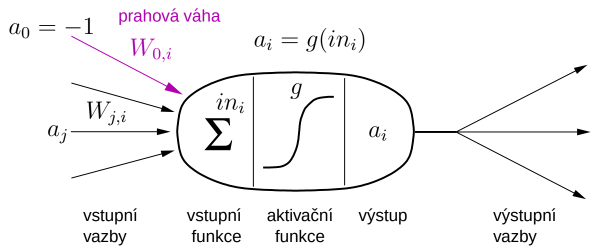
spojené do neuronové sítě – mají schopnost tolerovat šum ve vstupu a učit se

jednotky (*units*) v neuronové síti
- jsou propojeny vazbami (*links*)
- vazba z jednotky j do i propaguje aktivaci a_j jednotky j
- každá vazba má číselnou váhu $W_{j,i}$ (síla+znaménko)

funkce jednotky i:

1. spočítá váženou $\sum$ vstupů = in_i
2. aplikuje aktivační funkci g
3. tím získá výstup a_i

$$a_i = g(in_i) = \sum_j W_{j,i} a_j$$

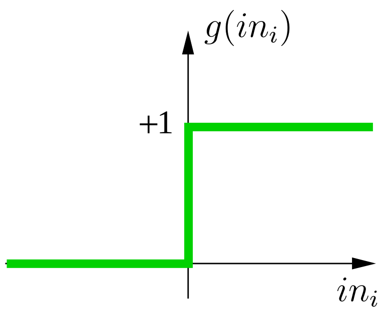


Aktivační funkce

účel aktivační funkce =
- jednotka má být aktivní ($\approx +1$) pro pozitivní příklady, jinak neaktivní ≈ 0
- aktivace musí být nelineární, jinak by celá síť byla lineární

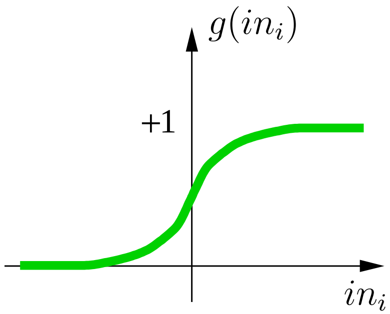
např.

a)



prahová funkce

b)

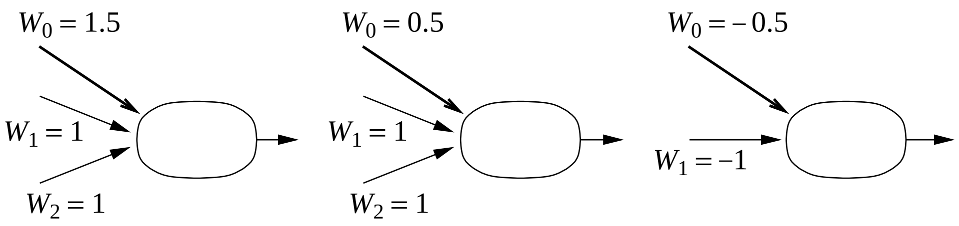


sigmoida $1/(1 + e^{-x})$

je derivovatelná – důležité pro učení

změny prahové váhy $W_{0,i}$ nastavují nulovou pozicí – nastavují práh aktivace

Logické funkce pomocí neuronové jednotky



AND **OR** **NOT**

jednotka McCulloch & Pitts sama umí implementovat základní Booleovské funkce

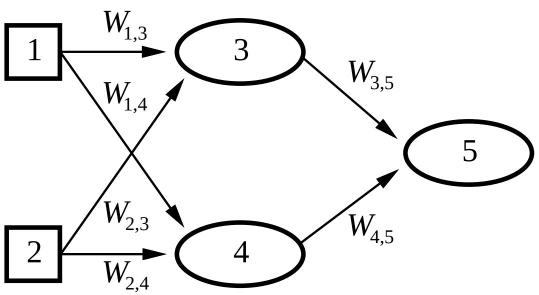
⇒ kombinacemi jednotek do sítě můžeme implementovat libovolnou Booleovskou funkci

Struktury neuronových sítí

- **sítě s předním vstupem** (*feed-forward networks*)
  - necyklické
  - implementují funkce
  - nemají vnitřní paměť
- **rekurentní sítě** (*recurrent networks*)
  - cyklické
  - vlastní výstup si berou opět na vstup
  - složitější a schopnější
  - výstup má (zpožděný) vliv na aktivaci = paměť
  - Hopfieldovy sítě – symetrické obousměrné vazby; fungují jako *asociativní paměť*
  - Boltzmannovy stroje – pravděpodobnostní aktivační funkce

Příklad sítě s předním vstupem

síť 5-ti jednotek – 2 vstupní jednotky, 1 skrytá vrstva (2 jednotky), 1 výstupní jednotka

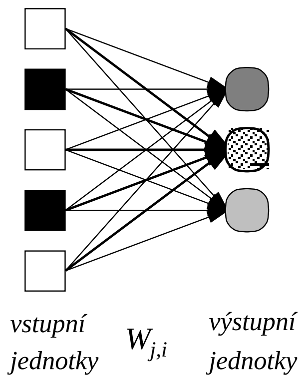


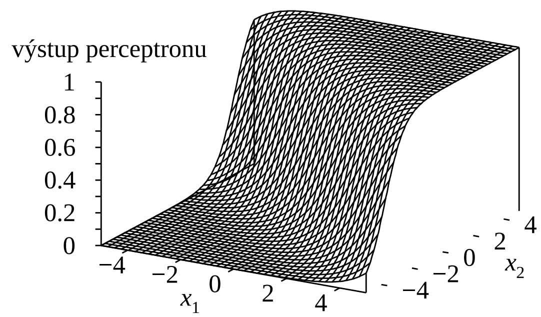
síť s předním vstupem = parametrizovaná nelineární funkce vstupu

$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g\big(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)\big) \end{aligned}$$

Jednovrstvá síť – perceptron

perceptron
- pro Booleovskou funkci 1 výstupní jednotka
- pro složitější klasifikaci – více výstupních jednotek





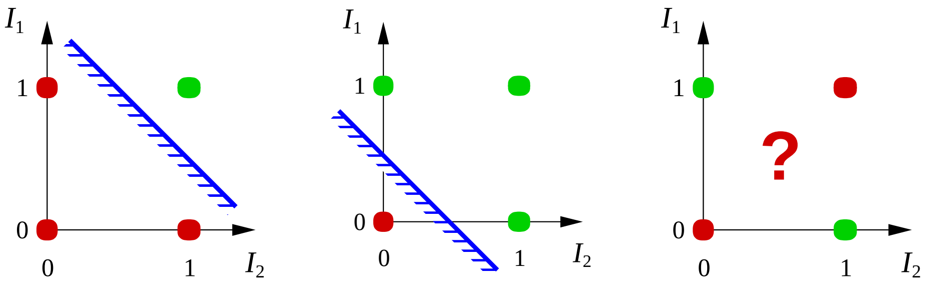
VYJADŘOVACÍ SÍLA PERCEPTRONU

předpokládejme perceptron s g zvolenou jako prahová funkce ( )

může reprezentovat hodně Booleovských funkcí – AND, OR, NOT, majoritní funkci, . . .

$$\sum_j W_j x_j > 0 \quad \text{nebo} \quad \mathbf{W} \cdot \mathbf{x} > 0$$

reprezentuje lineární separátor (nadrovina) v prostoru vstupu:



a) I_1 **and** I_2 b) I_1 **or** I_2 c) I_1 **xor** I_2

UČENÍ PERCEPTRONU

výhoda perceptronu – existuje jednoduchý učící algoritmus pro libovolnou lineárně separabilní funkci

učení perceptronu = upravování vah tak, aby se snížila chyba na trénovací sadě

kvadratická chyba E pro příklad se vstupem $\mathbf{x}$ a požadovaným (=správným) výstupem y je

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2, \qquad \text{kde } h_{\mathbf{W}}(\mathbf{x}) \text{ je (vypočítaný) výstup perceptronu}$$

váhy pro minimální chybu pak hledáme optimalizačním prohledáváním spojitého prostoru vah

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}\left(y - g(\textstyle\sum_{j=0}^{n} W_j x_j)\right) = -Err \times g'(in) \times x_j$$

pravidlo pro úpravu váhy $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j$ $\quad\alpha\ldots$učící konstanta (*learning rate*)

např. $Err = y - h_{\mathbf{W}}(\mathbf{x}) > 0 \quad \Rightarrow \quad$ výstup $h_{\mathbf{W}}(\mathbf{x})$ je moc malý

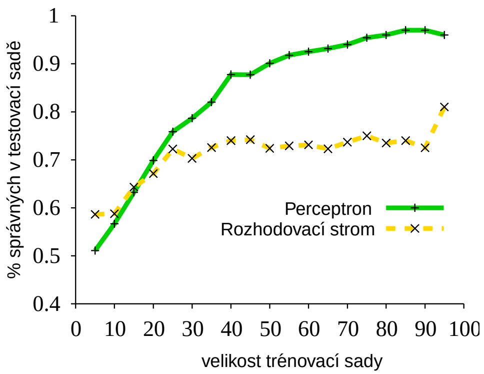
$\Rightarrow$ váhy se musí *zvýšit* pro pozitivní příklady a *snížit* pro negativní

úpravu vah provádíme po každém příkladu $\longrightarrow$ opakovaně až do dosažení ukončovacího kritéria

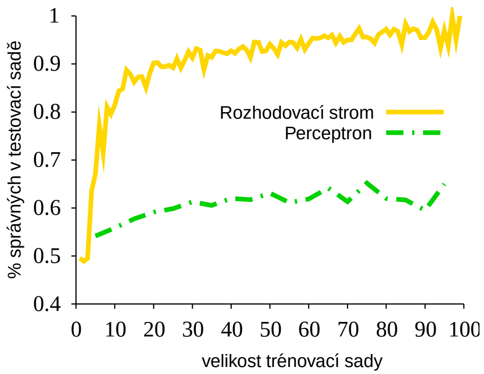
UČENÍ PERCEPTRONU pokrač.

učící pravidlo pro perceptron konverguje ke správné funkci pro libovolnou lineárně separabilní množinu dat

a) učení majoritní funkce



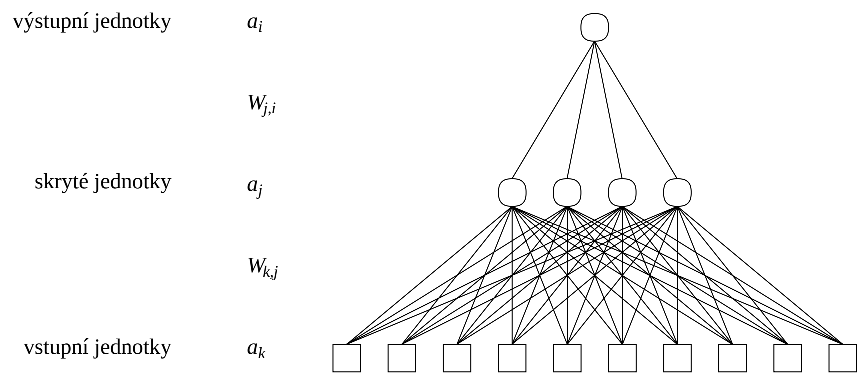
b) učení čekání na volný stůl v restauraci



VÍCEVRSTVÉ NEURONOVÉ SÍTĚ

vrstvy jsou obvykle úplně propojené

počet skrytých jednotek je obvykle volen experimentálně



Vyjadřovací síla vícevrstvých sítí

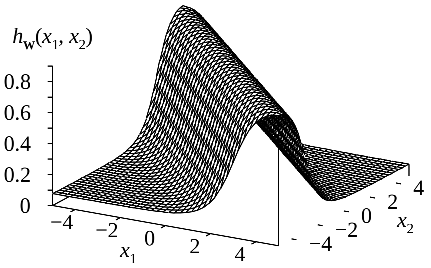
s jednou skrytou vrstvou – všechny spojité funkce

se dvěma skrytými vrstvami – všechny funkce

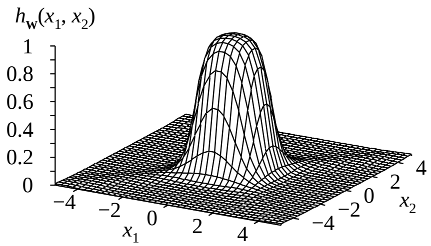
těžko se ovšem pro konkrétní síť zjišťuje její prostor reprezentovatelných funkcí

např.

dvě "opačné" skryté jednotky vytvoří *hřbet*



dva hřbety vytvoří *homoli*



Učení vícevrstvých sítí

pravidla pro úpravu vah:

- **výstupní vrstva** – stejně jako u perceptronu

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \qquad \text{kde} \quad \Delta_i = Err_i \times g'(in_i)$$

- **skryté vrstvy** – zpětné šíření (*back-propagation*) chyby z výstupní vrstvy

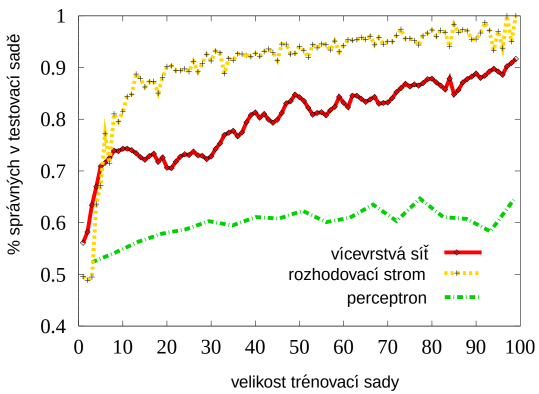
$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j \qquad \text{kde} \quad \Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i$$

problémy učení:

- dosažení lokálního minima chyby
- příliš pomalá konvergence
- přílišné upnutí na příklady $\longrightarrow$ neschopnost generalizovat

Učení vícevrstvých sítí pokrač.

vícevrstvá síť se problém čekání na volný stůl v restauraci učí znatelně líp než perceptron



Neuronové sítě – shrnutí

- ➜ většina mozků má velké množství neuronů; každý neuron ≈ lineární prahová jednotka (?)
- ➜ perceptrony (jednovrstvé sítě) mají nízkou vyjadřovací sílu
- ➜ vícevrstvé sítě jsou dostatečně silné; mohou být trénovány pomocí zpětného šíření chyby
- ➜ velké množství reálných aplikací
  - rozpoznávání řeči
  - řízení auta
  - rozpoznávání ručně psaného písma
  - ...

Reprezentace a vyvozování znalostí

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

➜ Reprezentace a vyvozování znalostí
➜ Logika
➜ Extralogické informace
➜ Rámce
➜ Pravidlové systémy
➜ Nejistota a pravděpodobnost

Reprezentace a vyvozování znalostí

otázka:

Jak zapíšeme znalosti o problému/doméně?
Když je zapíšeme, můžeme z nich mechanicky odvodit nová fakta?

➜ reprezentace znalostí (*knowledge representation*) – hledá způsob vyjádření znalostí počítačově zpracovatelnou formou (za účelem odvozování)

➜ vyvozování znalostí (*reasoning*) – zpracovává znalosti uložené v bázi znalostí (*knowledge base, KB*) a provádí odvození (inference) nových závěrů:
- odpovědi na dotazy
- zjištění faktů, které vyplývají z faktů a pravidel v KB
- odvodit akci, která vyplývá z dodaných znalostí, . . .

Reprezentace znalostí

proč je potřeba speciální reprezentace znalostí?

vnímání lidí $\times$ vnímání počítačů

člověk

➜ když dostane novou věc (třeba pomeranč) – prozkoumá a zapamatuje si ho (a třeba sní)
➜ během tohoto procesu člověk zjistí a uloží všechny základní vlastnosti
➜ později, když se zmíní daná věc, vyhledají se a připomenou uložené informace

počítač

➜ musí se spolehnout na informace od lidí
➜ jednodušší informace – přímé *programování*
➜ složité informace – zadané v symbolickém jazyce

Volba reprezentace znalostí

která reprezentace znalostí je nejlepší?

> To solve really hard problems, we'll have to use several different representations. This is because each particular kind of data structure has its own virtues and deficiencies, and none by itself would seem adequate for all the different functions involved with what we call common sense.
>
> – Marvin Minsky

Předpoklad uzavřeného světa

2 užitečné předpoklady:

- ➜ předpoklad uzavřeného světa (*closed world assumption*)
  - cokoliv o čem nevíme, že je pravda $\longrightarrow$ bereme za dané, že je to nepravda
  - využitý např. v Prologu (negace jako neúspěch)
- ➜ předpoklad jednoznačných pojmenování (*unique names assumption*)
  - různá jména označují různé objekty

Logika

vyvozování nových znalostí = hledání důkazu

algoritmus konstrukce důkazu:

- ➜ dopředné a zpětné řetězení – neúplné pro PL1
- ➜ rezoluce
- ➜ logické programování – SLD rezoluce

Historie logického vyvozování

450 př.n.l.	stoikové	výroková logika, inference (pravděpodobně)
322 př.n.l.	Aristoteles	inferenční pravidla, kvantifikátory
1565	Cardano	teorie pravděpodobnosti (výroková logika + nejistota)
1847	Boole	výroková logika (znovu)
1879	Frege	predikátová logika 1. řádu
1922	Wittgenstein	důkaz pomocí pravdivostních tabulek
1930	Gödel	$\exists$ úplný algoritmus pro PL1
1930	Herbrand	úplný algoritmus pro PL1 (redukce na výroky)
1931	Gödel	$\neg\exists$ úplný algoritmus pro aritmetiku
1960	Davis/Putnam	"practický" algoritmus pro výrokovou logiku
1965	Robinson	"practický" algoritmus for PL1 – rezoluce

Rezoluce v PL1

vyvozování v PL1 je pouze částečně rozhodnutelné:

- ➜ může najít důkaz α, když $KB \models \alpha$
- ➜ nemůže vždy dokázat, že $KB \not\models \alpha$
  viz *problém zastavení* – důkazová procedura nemusí skončit

rezoluce je důkaz sporem:

pro důkaz $KB \models \alpha$ ukážeme, že $KB \land \neg\alpha$ je nesplnitelné

rezoluce používá $KB, \neg\alpha$ v konjunktivní normální formě (CNF). Existuje přesný algoritmus pro převod každé PL1 klauzule do CNF, např.:

$$\begin{aligned}(P \lor Q) \Rightarrow (Q \Leftrightarrow R) \quad \equiv \quad & (\neg P \lor \neg Q \lor R) \\ \land \quad & (\neg P \lor Q \lor \neg R) \\ \land \quad & (\neg Q \lor R)\end{aligned}$$

Rezoluční pravidlo

algoritmus je založen na opakované aplikace rezolučního pravidla – ze dvou klauzulí odvoď novou klauzuli

$$\begin{tikzcd} C_1 \arrow[dr] & & C_2 \arrow[dl] \\ & C & \end{tikzcd}$$

- klauzule: $C_1 = P_1 \lor P_2 \lor \ldots \lor P_n$
  a $C_2 = \neg P_1 \lor Q_1 \lor Q_2 \lor \ldots \lor Q_m$
- výsledek: $C = P_2 \lor P_3 \lor \ldots \lor P_n \lor Q_1 \lor Q_2 \lor \ldots \lor Q_m$
- vyruší se opačné literály P_1 a $\neg P_1$

postup rezolučního důkazu tvrzení F:

- začneme s $\neg F$
- rezolvujeme s klauzulí z KB (která obsahuje F)
- opakujeme až do odvození prázdné klauzule $\Box$
- když se to podaří $\longrightarrow$ došli jsme ke sporu (pro $\neg F$) $\longrightarrow$ musí platit F

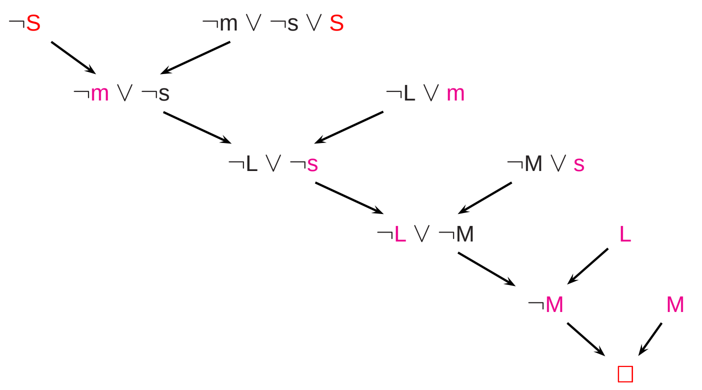
Rezoluce – příklad

- pravidla
  - mráz $\land$ srážky $\Rightarrow$ sněží
    $\neg$mráz $\lor$ $\neg$srážky $\lor$ sněží
  - Leden $\Rightarrow$ mráz
    $\neg$Leden $\lor$ mráz
  - mraky $\Rightarrow$ srážky
    $\neg$mraky $\lor$ srážky
- fakta – Leden, mraky
- dotaz (co se má dokázat) – sněží?

Důkaz tvrzení "sněží"

S – sněží, s – srážky, m – mráz, L – Leden, M – mraky

¬M ∨ ¬s ∨ S
¬L ∨ M
¬M ∨ s
L, M



Extralogické informace

co jsme dosud ignorovali:

- objekty reálného světa mají mezi sebou vztahy
  - třídy/kategorie, podtřídy × nadtřídy
  - hierarchie vztahů části/celku
  - dědění vlastností v hierarchiích
- stav světa se může měnit v čase
  - explicitní reprezentace času
  - nemonotónní uvažování (pravdivost se může měnit v čase)
- ne každá informace je "černobílá"
  - nejistota
  - statistika, fuzzy logika

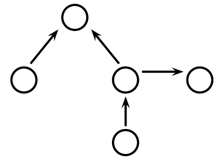
Třídy objektů

- “Chci si koupit fotbalový míč.”
  - *Chci si koupit FM27341* – špatně
  - *Chci si koupit objekt, který je prvkem třídy fotbalových míčů* – správně
- objekty jsou organizovány do hierarchie tříd
  - FM27341 $\in$ fotbalové_míče
  - fotbalové_míče $\subset$ míče
- fakta (objekty) $\times$ pravidla (třídy)
  - Všechny míče jsou kulaté.
  - Všechny fotbalové míče mají X cm v průměru.
  - FM27341 je červenomodrobílý.
  - FM27341 je fotbalový míč.
  - (Proto: FM27341 je kulatý a má X cm v průměru.)

Sémantické sítě

sémantické sítě – reprezentace faktových znalostí (pojmy + vztahy)

- vznikly kolem roku 1960 pro reprezentaci významu anglických slov
- znalosti jsou uloženy ve formě grafu



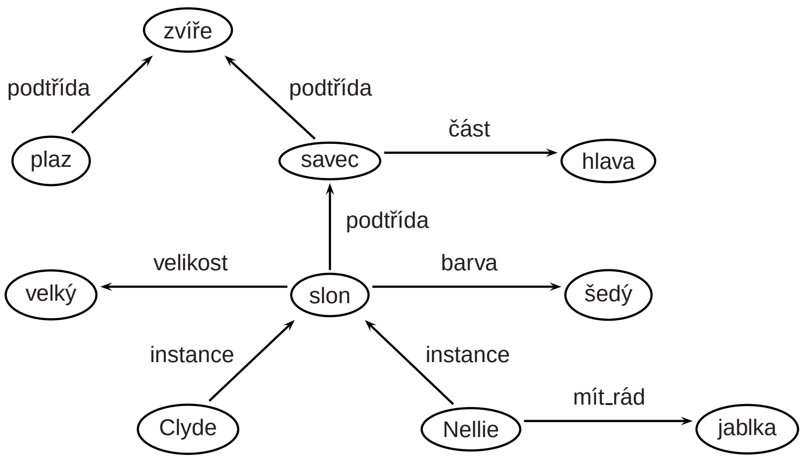
○ pojmy (objekty, třídy)

$\longrightarrow$ vztahy

- nejdůležitější vztahy:
  - podtřída (*subclass*) – vztah mezi třídami
  - instance – vztah mezi konkrétním objektem a jeho rodičovskou třídou

  jiné vztahy – část (has-part), barva, . . .

Sémantické sítě – příklad



Dědičnost v sémantických sítích

- pojem sémantické sítě *předchazí* OOP
- dědičnost:
  - jestliže určitá vlastnost platí pro třídu$\longrightarrow$ platí i pro všechny její podtřídy
  - jestliže určitá vlastnost platí pro třídu$\longrightarrow$ platí i pro všechny prvky této třídy
- určení hodnoty vlastnosti – rekurzivní algoritmus
- potřeba specifikovat i výjimky – mechanizmus vzorů a výjimek (*defaults and exceptions*)
  - vzor – hodnota vlastnosti u třídy nebo podtřídy, platí ta, co je blíž objektu
  - výjimka – u konkrétního objektu, odlišná od vzoru

DĚDIČNOST VZTAHŮ ČÁST/CELEK

- “krávy mají 4 nohy.”
  - každá noha je částí krávy
- “Na poli je (konkrétní) kráva.”
  - všechny části krávy jsou taky na poli
- “Ta kráva (na poli) je hnědá (celá).”
  - všechny části té krávy jsou hnědé
- “Ta kráva je šťastná.”
  - ~~všechny části té krávy jsou šťastné~~ – neplatí
- lekce: některé vlastnosti jsou děděny částmi, některé nejsou
  explicitně se to vyjadřuje pomocí pravidel jako
  part-of$(x, y) \wedge$ location$(y, z) \Rightarrow$ location(x, z)

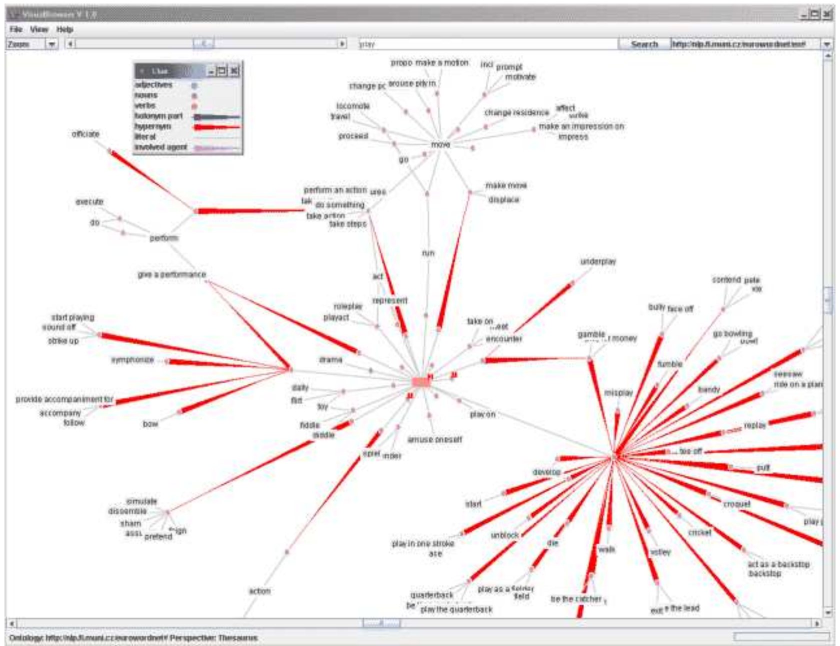
VZORY A VÝJIMKY – PŘÍKLAD

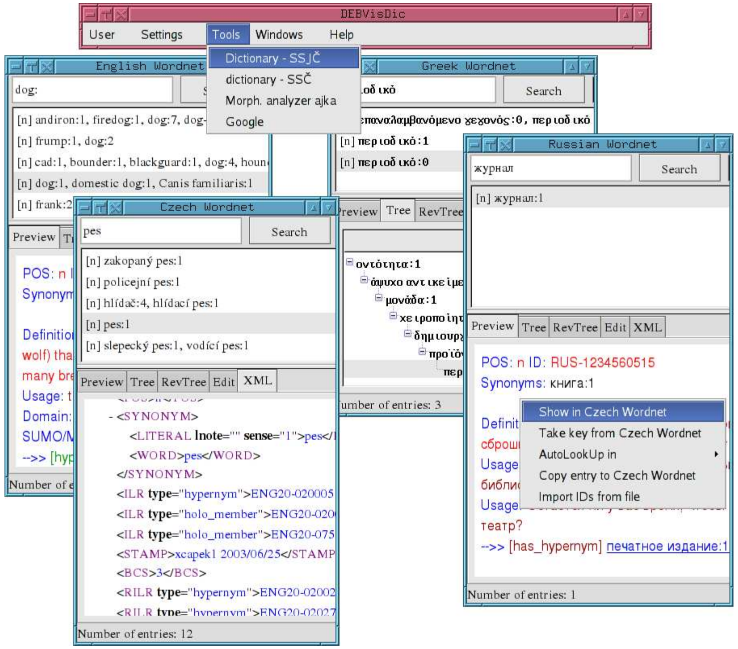
- “všichni ptáci mají křídla.”
- “všichni ptáci umí létat.”
- “ptáci se zlomenými křídly jsou ptáci, ale neumí létat.”
- “tučnáci jsou ptáci, ale neumí létat.”
- “kouzelní tučňáci jsou tučňáci, kteří umí létat.”
- kdo umí létat:
  - “Tweety je pták.”
  - “Petřík je tučnák.”
  - “Penelope je kouzelný tučnák.”
- všimněte si, že víra v hodnotu vlastnosti objektu se může měnit s příchodem nových informací o klasifikaci objektu

APLIKACE SÉMANTICKÝCH SÍTÍ

(Princeton) WordNet – `http://wordnet.princeton.edu/`

- sémantická síť 100.000 (anglických) pojmů, zachycuje:
  - synonyma, antonyma (významově stejná/opačná)
  - hyperonyma, hyponyma (podtřídy)
  - odvozenost a další jazykové vztahy
- tvoří se národní wordnety (navázané na anglický WN)
  český wordnet – cca 30.000 pojmů
- nástroj na editaci národních wordnetů – DEBVisDic/VisDic, vyvinutý na FI MU –
  `http://nlp.fi.muni.cz/projekty/deb2/`
- VisualBrowser – `http://nlp.fi.muni.cz/projekty/visualbrowser/`
  nástroj na vizualizaci (sémantických) sítí, vznikl jako DP na FI MU





Rámce

Rámce (*frames*):

➜ varianta sémantických sítí

➜ velice populární pro reprezentaci znalostí v expertních systémech

➜ všechny informace relevantní pro daný pojem se ukládají do univerzálních struktur – rámců

➜ stejně jako sémantické sítě, rámce podporují dědičnost

➜ OO programovací jazyky vycházejí z teorie rámců

Rámce – příklad

rámec obsahuje objekty, *sloty* a hodnoty slotů

příklady rámců:

savec:

podtřída:	zvíře
část:	hlava
**má_kožich*:	ano

slon:

podtřída:	savec
**barva*:	šedá
**velikost*:	velký

Nellie:

instance:	slon
mít_rád:	jablka

'*' označuje vzorové hodnoty, které mohou měnit hodnoty u podtříd a instancí

Sémantické sítě × rámce

sémantické sítě	rámce
uzly	objekty
spoje	sloty
uzel na druhém konci spoje	hodnota slotu

deskripční logika – logický systém, který manipuluje přímo s rámci

Pravidlové systémy

➜ snaha zachytit produkčními pravidly znalosti, které má expert

➜ obecná forma pravidel

IF podmínka

THEN akce

– podmínky – booleovské výrazy, dotazy na hodnoty proměnných
– akce – nastavení hodnot proměnných, příznaků, ...

➜ důležité vlastnosti:

– znalosti mohou být strukturovány do modulů
– systém může být snadno rozšířen přidáním nových pravidel beze změny zbytku systému

Pravidlová báze znalostí – příklad

pravidla pro oblékání:

pravidlo 1 IF X je seriózní
AND X bydlí ve městě
THEN X by měl nosit sako

pravidlo 2 IF X je akademik
AND X je společensky aktivní
AND X je seriózní
THEN X by měl nosit sako a kravatu

pravidlo 3 IF X bydlí ve městě
AND X je akademik
THEN X by měl nosit kravatu

pravidlo 4 IF X je podnikatel
AND X je společensky aktivní
AND X je seriózní
THEN X by měl nosit sako, ale ne kravatu

společenská pravidla:

pravidlo 5 IF X je podnikatel
AND X je ženatý
THEN X je společensky aktivní

pravidlo 6 IF X je akademik
AND X je ženatý
THEN X je seriózní

profesní pravidla:

pravidlo 7 IF X učí na univerzitě
OR X učí na vysoké škole
THEN X je akademik

pravidlo 8 IF X vlastní firmu
OR X je OSVČ
THEN X je podnikatel

Expertní systémy

➜ aplikace pravidlových systémů

➜ zaměřeny na specifické oblasti – medicínská diagnóza, návrh konfigurace počítače, expertíza pro těžbu nafty, ...

➜ snaha zachytit znalosti experta pomocí pravidel
ale znalosti experta zahrnují – postupy, strategie, odhady, ...

➜ expertní systém musí pracovat s procedurami, nejistými znalostmi, různými formami vstupu

➜ vhodné oblasti pro nasazení expertního systému:

– diagnóza – hledání řešení podle symptomů
– návrh konfigurace – složení prvků splňujících podmínky
– plánování – posloupnost akcí splňujících podmínky
– monitorování – porovnání chování s očekávaným chování, reakce na změny
– řízení – ovládání složitého komplexu
– předpovědi – projekce pravěpodobných závěrů z daných skutečností
– instruktáž – inteligentní vyučování a zkoušení studentů

Nejistota

definujme akci A_t jako "Vyrazit na letiště t hodin před odletem letadla."
jak najít odpověď na otázku "*Dostanu se akcí A_t na letiště včas k odletu letadla?*"

problémy:

1. částečná pozorovatelnost (stav vozovky, záměry ostatních řidičů, ...)
2. šum v senzorech (hlášení o dopravní situaci)
3. nejistota výsledků akcí (píchnutí kola, ...)
4. obrovská složitost modelování a předpovědi dopravní situace

čistě logický přístup tedy:

– riskuje chybu – "A_5 mě tam dostane včas."
– vede k závěrům, které jsou příliš slabé pro rozhodování: "A_5 mě tam dostane včas, pokud nebude na dálnici nehoda a pokud nebude pršet a jestli nepíchnu kolo a jestli nebude fronta na odbavovacích přepážkách a jestli nebudou problémy při kontrole zavazadel ..."

Metody pro práci s nejistotou

defaultní/nemonotónní logika

Předpokládejme, že nepíchnu cestou kolo.

Předpokládejme, že A_5 bude OK, pokud se nenajde protipříklad.

pravidla s faktory nejistoty

$A_5 \mapsto_{0.3}$ dostat se na letiště včas.

zalévání $\mapsto_{0.99}$ mokrý trávník

mokrý trávník $\mapsto_{0.7}$ déšť

pravděpodobnost

Vzhledem k dostupným informacím, A_3 mě tam dostane včas s pravděpodobností 0.05.

poznámka: fuzzy logika se zabývá mírou pravdivosti, NE nejistotou

Pravděpodobnost

tvrzení o pravděpodobnosti shrnují následky

- lenosti – nepodařilo se vypočítat všechny výjimky, podmínky, ...
- neznalosti – nedostatek relevantních údajů, počátečních podmínek, ...

(takže přesně popisují běžnou práci v IT ☺)

subjektivní × Bayesovská pravděpodobnost:

- pravděpodobnostní vztah mezi tvrzením a jeho pravdivosti vzhledem k podmínkám:

  $P(A_4|\text{žádné hlášené nehody}) = 0.5$

  nejedná se o vyjádření pravděpodobnostní tendence (ale může se získat ze znalostí podobných případů v minulosti)
- pravděpodobnost tvrzení se může měnit s novými (vstupními) podmínkami:

  $P(A_4|\text{žádné hlášené nehody, je 4:00 ráno}) = 0.63$

Vyvozování z nejistých znalostí

- ➜ použití náhodných proměnných (*random variables*) – funkce, která vzorkům přirazuje hodnoty $\longrightarrow$ vrací výsledky měření sledovaného jevu

  distribuce pravděpodobností náhodné proměnné = (vektor) pravděpodobnost(í), že daná náhodná proměnná bude mít určitou konkrétní hodnotu

  např.: náhodná proměnná Odd vyjadřující, že výsledek hodu kostkou bude lichý

  náhodná proměnná $Weather$ vyjadřující, jaké bude počasí (slunce, déšť, mraky, sníh)

  $$Odd(1) = true \qquad Weather(\text{21.11.2005}) = \text{déšť}$$

  distribuce pravděpodobností proměnných Odd a $Weather$

  $$P(Odd = true) = 1/6 + 1/6 + 1/6 = 1/2$$
  $$P(Odd) =< 1/2, 1/2 >$$
  $$P(Weather) =< 0.72, 0.1, 0.08, 0.1 >$$

- ➜ pravidla pro výpočet pravděpodobnosti logicky souvisejících událostí

  $$P(a \vee b) = P(a) + P(b) - P(a \wedge b)$$

Bayesovské pravidlo pro vyvozování

pravidlo pro podmíněnou pravděpodobnost – $P(a|b) = \frac{P(a \wedge b)}{P(b)}$ if $P(b) \neq 0$

Bayesovské pravidlo pro určení diagnostické pravděpodobnosti ze znalosti příčinné pravděpodobnosti:

$$P(\text{Příčina}|\text{Následek}) = \frac{P(\text{Následek}|\text{Příčina})P(\text{Příčina})}{P(\text{Následek})}$$

např. ZMB zánět mozkových blan, ZK ztuhlý krk:

$$P(zmb|zk) = \frac{P(zk|zmb)P(zmb)}{P(zk)} = \frac{0.8 \times 0.0001}{0.1} = 0.0008$$

vyvozování =

1. rozdělení akce na atomické události
2. zjištění pravděpodobností atomických událostí
3. výpočet/odvození pravděpodobností pomocí složených distribucí pravděpodobností (*joint probability distribution*)

Zpracování přirozeného jazyka

Aleš Horák

E-mail: `hales@fi.muni.cz`

`http://nlp.fi.muni.cz/uui/`

Obsah:

➜ Zpracování přirozeného jazyka
➜ DC gramatiky – gramatiky uspořádaných klauzulí
➜ Analýza přirozeného jazyka
➜ Syntaktická analýza přirozeného jazyka

PŘIROZENÝ JAZYK – PROSTŘEDEK KOMUNIKACE

komunikace = cílená výměna informace pomocí produkce a vnímání (sdílených) pokynů

- zvířata – až stovky pokynů (šimpanz, delfín, ...)
- člověk – potenciálně neomezené množství, díky přirozenému jazyku

2 náhledy na přirozený jazyk:

❑ **klasický (před 1953)** – jazyk se skládá z vět, které jsou buď pravdivé nebo nepravdivé (srovnej s logikou)

❑ **moderní (po 1953)** – užití jazyka je jedna z možných akcí

Wittgenstein (1953) Philosophical Investigations
Searle (1969) Speech Acts

Turingův test založen na jazyku $\Leftarrow$ jazyk je pevně spojen s myšlením

komunikace se tvoří pomocí řečových aktů (*speech acts*) jako jeden z typů agentových akcí

cíl komunikace – změnit akce ostatních agentů

ŘEČOVÉ AKTY

SITUACE
Mluvčí (*speaker*) ⟶ Promluva (*utterance*) ⟶ Posluchač (*hearer*)

řečové akty směřují k naplnění cílů mluvčího:

– informovat (inform)	“Před tebou je jáma.”
– ptát se (query)	“Vidíš zlato?”
– přikázat/žádat (command/request)	“Zvedni to.”
– slíbit/svěřit se s plánem (promise, commit to plan)	“Rozdělím se s tebou o zlato.”
– potvrdit (acknowledge)	“OK”

plánování řečových aktů vyžaduje znalosti:

- situace
- sémantiky a syntaxe (sdílených konvencí)
- informace o Posluchači – cíle, znalosti, rozumnost

KOMUNIKAČNÍ FÁZE (PŘI INFORMOVÁNÍ)

průběh promluvy je možné rozložit na fáze:

– záměr (intention)	M chce informovat Po, že Pr
– generování (generation)	M vybírá slova W pro vyjádření Pr
– syntéza (synthesis)	M říká slova W
– vnímání (perception)	Po vnímá W'
– analýza (analysis)	Po odvozuje možné významy $Pr_1, \ldots, Pr_n$
– zjednoznačnění (disambiguation)	Po vybírá zamýšlený význam Pr_i
– zahrnutí (incorporation)	Po zahrne Pr_i do své báze znalostí

Může přitom vzniknout chyba?

- neupřímnost (Po nevěří Pr)
- víceznačnost promluvy (Po zvolí špatné Pr_i)
- různé pochopení aktuální situace (zamýšlený význam mezi Pr_i není)

Komunikační fáze – příklad

Mluvčí

záměr
Vědět$(Po, \neg$Na_živu$(Wumpus_1, S_3))$

generování
"Wumpus je mrtvý."

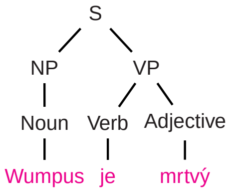
syntéza
[v u m p u s j e m r t v i:]

Posluchač

vnímání
"Wumpus je mrtvý."

analýza

syntaktická analýza:



sémantická interpretace:
$\neg$Na_živu$(Wumpus,$ Teď$)$
$\neg$Unavený$(Wumpus,$ Teď$)$

pragmatická interpretace:
$\neg$Na_živu$(Wumpus_1, S_3)$
$\neg$Unavený$(Wumpus_1, S_3)$

zjednoznačnění
$\neg$Na_živu$(Wumpus, S_3))$

zahrnutí
Tell$(KB, \neg$Na_živu$(Wumpus, S_3))$

Gramatika

zvířata používají místo vět izolované symboly $\Rightarrow$ omezená sada komunikovatelných situací
$\longrightarrow$ žádná generativní kapacita

gramatika specifikuje skladební strukturu složených pokynů – definuje *formální jazyk* pokynů

formální jazyk = množina řetězců (vět) teminálních symbolů (slov)

2 náhledy na vztah věty a gramatiky:

- S je správný řetězec/věta z jazyka $\Leftrightarrow$ S je analyzovatelný příslušnou gramatikou
- příslušná gramatika generuje S $\Leftrightarrow$ S je správný řetězec/věta z jazyka

gramatika je zadána jako množina přepisovacích pravidel, např.

$S \rightarrow NP\ VP$
$Pronoun \rightarrow$ já | ty | on | ...

v tomto příkladu:

S	větný symbol – kořenový symbol gramatiky
NP, VP	neterminály
já, ty, ...	terminály

Typy gramatik

gramatiky:

- **regulární** (regular) neterminál $\longrightarrow$ **terminál**[neterminál]

  S $\longrightarrow$ **a**S
  S $\longrightarrow$ **b**

  ekvivalentní síle konečných automatů, neumí $a^n b^n$
- **bezkontextové** (context-free) neterminál $\longrightarrow$ cokoliv

  S $\longrightarrow$ **a**S**b**

  ekvivalentní síle zásobníkových automatů, umí $a^n b^n$, neumí $a^n b^n c^n$
- **kontextové** (context-sensitive) – víc neterminálů na levé straně; na levé straně se jejich počet "zmenšuje"

  ASB $\longrightarrow$ AA**a**BB

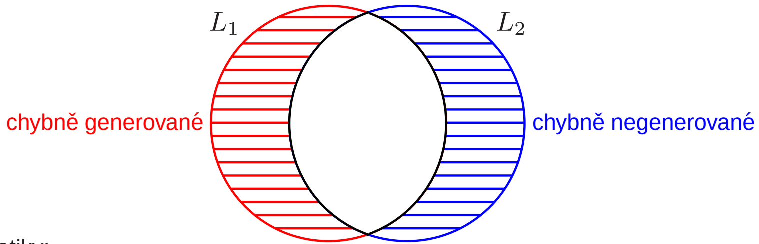
  umí $a^n b^n c^n$
- **rekurzivně vyčíslitelné** (recursively enumerable) – bez omezení

  ekvivalentní síle Turingova stroje

přirozený jazyk byl dlouho pokládán za bezkontextový $\longrightarrow$ nyní prokázáno, že obsahuje kontextové prvky

Přesnost a pokrytí gramatiky

u složitějších jazyků (např. přirozených)
$\longrightarrow$ jazyk L_1 (generovaný gramatikou) se liší od zamýšleného jazyka L_2



kvalita gramatiky:

- pokrytí – procento vět jazyka L_2 generovatelných gramatikou $(|L_1 \cap L_2|/|L_2|)$
- přesnost – procento generovaných vět, které jsou správné věty jazyka L_2 $(|L_1 \cap L_2|/|L_1|)$

tvorba gramatiky ... postupný proces zvyšování pokrytí a přesnosti

gramatiky přirozených jazyků – velmi rozsáhlé a přesto většinou nepopisují plně ani angličtinu ☹

DC gramatiky – gramatiky uspořádaných klauzulí

- *Definite-Clause Grammars*, DCG
- významná aplikace Prologu – *syntaktická analýza*
- DCG jsou rozšířením bezkontextových gramatik (CFG)
- jejich implementace využívá *rozdílových seznamů*

Formální podobnosti mezi DCG a CFG:

- CFG: pravidla tvaru $x \longrightarrow y$, kde $x \in N$ je neterminál a $y \in (N \cup T)^*$ je konečná posloupnost terminálů a neterminálů
- DCG: pravidla tvaru ⟨**hlava**⟩ **-->** ⟨**tělo**⟩, kde ⟨**hlava**⟩ je opět neterminál a ⟨**tělo**⟩ je opět konečná posloupnost terminálů a neterminálů
- pravidlo ⟨**hlava**⟩ **-->** ⟨**tělo**⟩ znamená, že jedním z možných tvarů ⟨**hlavy**⟩ je **tělo**, neboli: ⟨**hlavu**⟩ je možno přepsat na ⟨**tělo**⟩

Rozdíly a rozšíření DCG oproti CFG

1. Neterminál může být téměř libovolný term, kromě *seznamu*, *proměnné* a *čísla*.
2. Terminál může být libovolný term, s tím, že terminály a posloupnosti terminálů uzavíráme do hranatých závorek – jako seznamy.
3. Pravá strana pravidla může obsahovat dodatečné podmínky v podobě prologovských podcílů. Tyto podmínky uzavíráme do složených závorek.
4. Levá strana pravidla může dokonce vypadat i tak, že neterminál je následován posloupností terminálů.
5. Tělo pravidla smí obsahovat řez.

DC gramatika – příklad 1

gramatika vět typu “The young boy sings a song.”

```
% 1. část -- pravidla
sentence --> noun_phrase, verb_phrase.

noun_phrase --> determiner, noun_phrase2.
noun_phrase --> noun_phrase2.

noun_phrase2 --> adjective, noun_phrase2.
noun_phrase2 --> noun.

verb_phrase --> verb.
verb_phrase --> verb, noun_phrase.

% 2. část -- lexikon
determiner --> [the].        noun --> [boy].
determiner --> [a].          noun --> [song].

verb --> [sings].            adjective --> [young].
```


Analýza v Prologu pomocí append

- větu reprezentujeme seznamem slov **[the,young,boy,sings,a,song]**
- pravidlová část – neterminál chápeme jako unární predikát, jehož argumentem je ta větná složka, kterou daný neterminál popisuje

```
sentence(S) :- append(NP,VP,S),
               noun_phrase(NP), verb_phrase(VP).
...
```

- slovníková část, lexikon – zapisujeme pomocí faktů:

```
determiner([the]).        noun([boy]).
determiner([a]).          ...
```

Efektivněji – rozdílové seznamy

přepis gramatiky do Prologu pomocí rozdílových seznamů:

```
sentence(S,S0) :- noun_phrase(S,S1), verb_phrase(S1,S0).

noun_phrase(S,S0) :- determiner(S,S1), noun_phrase2(S1,S0).
noun_phrase(S,S0) :- noun_phrase2(S,S0).
noun_phrase(S,S0) :- adjective(S,S1), noun_phrase2(S1,S0).
noun_phrase2(S,S0) :- noun(S,S0).
verb_phrase(S,S0) :- verb(S,S0).
verb_phrase(S,S0) :- verb(S,S1), noun_phrase(S1,S0).

determiner([the|S],S).          noun([boy|S],S).
determiner([a|S],S).            noun([song|S],S).
verb([sings|S],S).              adjective ([young|S],S).

?- sentence([the,young,boy,sings,a,song],[]).
     Yes
```


Lexikon pro agenta ve Wumpusově jeskyni

rozdělení slov do kategorií:

podst. jméno:	*Noun*	$\longrightarrow$	zápach | vánek | třpyt | nic | wumpuse | jáma | zlato | ...
sloveso:	*Verb*	$\longrightarrow$	jsem | je | vidím | cítím | působí | zapáchá | jdu | ...
příd. jméno:	*Adjective*	$\longrightarrow$	levý | pravý | východní | jižní | ...
příslovce:	*Adverb*	$\longrightarrow$	tady | tam | blízko | vpředu | vpravo | vlevo | východně | jižně | vzadu | ...
vl. jméno:	*Name*	$\longrightarrow$	Petr | Honza | Brno | FI MU | ...
zájmeno:	*Pronoun*	$\longrightarrow$	já | ty | mě | toho | ten | ta ...
předložka:	*Preposition*	$\longrightarrow$	do | v | na | u | ...
spojka:	*Conjunction*	$\longrightarrow$	a | nebo | ale | ...
číslice:	*Digit*	$\longrightarrow$	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

kategorie můžeme dělit na otevřené (vyvíjející se) a uzavřené (stálé)

Morfologická analýza

- V češtině u lexikonu nestačí prostý výčet tvarů – je nutná morfologická analýza (morfologie=tvarosloví)
- skloňovaná a časovaná slova se rozkládají na segmenty

  pří-lež-it-ost-n-ými

  pří – prefix; *lež* – kořen; *it*, *ost*, *n* – suffixy; *ými* – koncovka
- každé slovo má základní tvar (*lemma*), podle koncovky se určují gramatické kategorie

```
% slovník základních gramatických kategorií –– pád, číslo, rod
% adj(+Slovo, +Lemma, +Pad, +Cislo, +Rod)
adj(chytrý, chytrý, 1, sg, mz).    adj(chytrého, chytrý, 2, sg, mz).    adj(chytří, chytrý, 1, pl, mz).
```

- reálná morfologická analýza ČJ – program Ajka na FI MU

```
ajka>nejneuvěřitelněji
<s> nej-ne=uvěřiteln==ěji= (1022)
    <l>uvěřitelně
    <c>k6xMeNd3
```

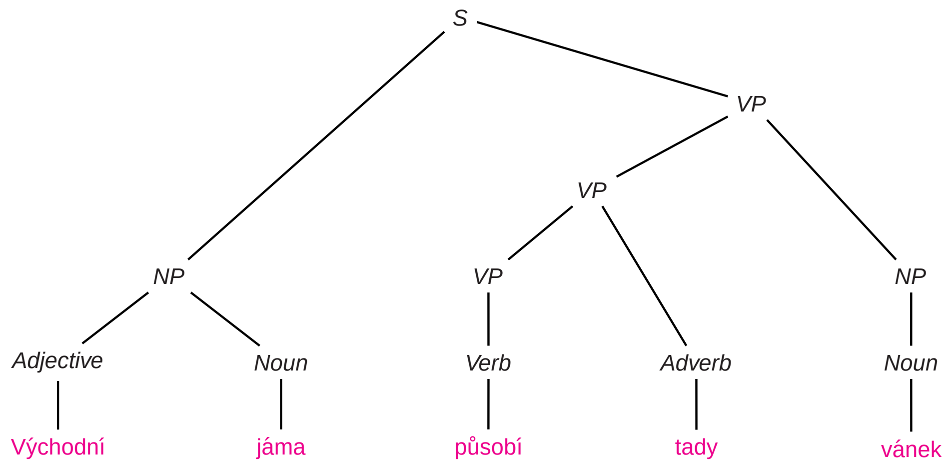
```
ajka>hnát
<s> ==hná=t= (618)
    <l>hnát
    <c>k5eAmFaI
<s> =hnát=== (1030)
    <l>hnát
    <c>k1gInSc1
    <c>k1gInSc4
```


Gramatická pravidla pro agenta ve Wumpusově jeskyni

S	$\longrightarrow$	*NP VP*	% já + cítím vánek
	|	*S Conjunction S*	% já cítím vánek + a + já jdu na východ
NP	$\longrightarrow$	*Pronoun*	% já
	|	*Noun*	% jáma
	|	*Adjective Noun*	% levá jáma
	|	*Pronoun NP*	% toho + wumpuse
	|	*Noun Digit* ',' *Digit*	% pole + 3,4
	|	*NP PP*	% jáma + na východě
	|	*NP RelClause*	% toho wumpuse + ,který zapáchá
VP	$\longrightarrow$	*Verb*	% zapáchá
	|	*VP NP*	% cítím + vánek
	|	*VP Adjective*	% je + třpytivý
	|	*VP PP*	% jdu + na východ
	|	*VP Adverb*	% jdu + dopředu
PP	$\longrightarrow$	*Preposition NP*	% na + východ
RelClause	$\longrightarrow$	', který' *VP*	% ,který + zapáchá

Syntaktický strom

syntaktický strom vzniká během syntaktické analýzy a dává záznam o jejím průběhu:



Konstrukce derivačního stromu

Neterminály opatříme argumentem:

```
sentence(sentence(NP,VP)) −−> noun_phrase(NP), verb_phrase(VP).
```

Převod do podoby klauzulí:

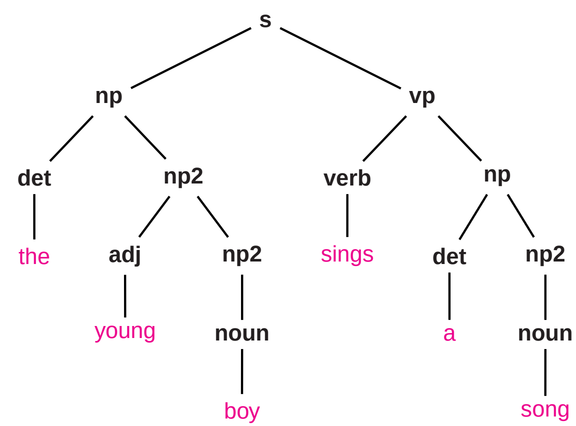
```
sentence(sentence(NP,VP),S,S0) :− noun_phrase(NP,S,S1), verb_phrase(VP,S1,S0).
```


DC gramatika s konstrukcí stromu analýzy

```
sentence(s(N,V)) −−> noun_phrase(N), verb_phrase(V).
noun_phrase(np(D,N)) −−> determiner(D), noun_phrase2(N).
noun_phrase(np(N)) −−> noun_phrase2(N).
noun_phrase2(np2(A,N)) −−> adjective(A), noun_phrase2(N).
noun_phrase2(np2(N)) −−> noun(N).
verb_phrase(vp(V)) −−> verb(V).
verb_phrase(vp(V,N)) −−> verb(V), noun_phrase(N).

determiner(det(the)) −−> [the].
determiner(det(a)) −−> [a].
adjective (adj(young)) −−> [young].
noun(noun(boy)) −−> [boy].
noun(noun(song)) −−> [song].
verb(verb(sings)) −−> [sings].

?− sentence(Tree, [the,young,boy,sings,a,song],[]).
     Tree=s(np(det(the),np2(adj(young),np2(noun(boy)))),
               vp(verb(sings),np(det(a),np2(noun(song)))))
```


Derivační strom analýzy v DC gramatikách

```
?− sentence(Tree, [the, young, boy, sings, a, song], []).
    Tree=s(np(det(the), np2(adj(young), np2(noun(boy)))),
              vp(verb(sings), np(det(a), np2(noun(song)))))
```



Test na shodu

Pokud však rozšíříme slovník:

```
noun(noun(boys)) --> [boys].
verb(verb(sing)) --> [sing].
```

Narazíme na problém se shodou v čísle:

```
?- sentence(_,[a, young, boys, sings ],[]).
Yes

?- sentence(_,[a, boy, sing ],[]).
Yes
```

Proto rozšíříme neterminály o další argument **Num**, ve kterém můžeme testovat shodu:

```
sentence(sentence(NP,VP)) --> noun_phrase(NP,Num), verb_phrase(VP,Num).
```


DC gramatika s testy na shodu

```
sentence(sentence(N,V)) --> noun_phrase(N,Num), verb_phrase(V,Num).
noun_phrase(np(D,N),Num) --> determiner(D,Num), noun_phrase2(N,Num).
noun_phrase(np(N),Num) --> noun_phrase2(N,Num).
noun_phrase2(np2(A,N),Num) --> adjective(A), noun_phrase2(N,Num).
noun_phrase2(np2(N),Num) --> noun(N,Num).
verb_phrase(vp(V),Num) --> verb(V,Num).
verb_phrase(vp(V,N),Num) --> verb(V,Num), noun_phrase(N,Num1).

determiner(det(the),_) --> [the].      noun(noun(boy),sg) --> [boy].
determiner(det(a),sg) --> [a].         noun(noun(song),sg) --> [song].
verb(verb(sings),sg) --> [sings].      noun(noun(boys),pl) --> [boys].
verb(verb(sing),pl) --> [sing].        noun(noun(songs),pl) --> [songs].
adjective(adj(young)) --> [young].

?- sentence(_,[a, young, boys, sings ],[]).
   No
?- sentence(_,[the,boys,sings,a,song ],[]).
   No
?- sentence(_,[the,boys,sing,a,song ],[]).
   Yes
```


Podmínky v těle pravidel

DC gramatiky mohou mít pomocné podmínky v těle pravidel – libovolný Prologovský kód

např. CFG pro vyhodnocení artimetického výrazu:

$$E \rightarrow T + E \mid T - E \mid T$$
$$T \rightarrow F * T \mid F/T \mid F$$
$$F \rightarrow (E) \mid f$$

zapíšeme včetně výpočtu hodnoty výrazu:

```
expr(X) --> term(Y), [+], expr(Z), {X is Y+Z}.
expr(X) --> term(Y), [-], expr(Z), {X is Y-Z}.
expr(X) --> term(X).

term(X) --> factor(Y), [*], term(Z), {X is Y*Z}.
term(X) --> factor(Y), [/], term(Z), {X is Y/Z}.
term(X) --> factor(X).

factor(X) --> ['('], expr(X), [')'].
factor(X) --> [X], {integer(X)}.

?- expr(X,[3,+,4,/,2,-,'(',2,*,6,/,3,+,2,')'],[]).
   X = -1
```


Generativní síla DCG

Generativní (rozpoznávací) síla DCG je větší než CFG

např. jazyk $a^n b^n c^n$:

```
abc --> a(N), b(N), c(N).

a(0) --> [].
a(s(N)) --> [a], a(N).

b(0) --> [].
b(s(N)) --> [b], b(N).

c(0) --> [].
c(s(N)) --> [c], c(N).

?- abc(X,[]).
X = [] ;
X = [a, b, c] ;
X = [a, a, b, b, c, c] ;
X = [a, a, a, b, b, b, c, c, c] ;
...
```

Význam syntaktické analýzy

- ➜ analýza syntaxe je nutná pro analýzu významu
- ➜ většina teorií analýzy významu dodržuje princip kompozicionality:

  Význam složeného výrazu je funkcí významu jednotlivých podvýrazů

- ➜ proces sémantické analýzy:
  - buď vychází z výsledků syntaktické analýzy
  - nebo probíhá současně se syntaktickou analýzou; pak může zasahovat i do tvorby syntaktického stromu

Problémy při analýze přirozeného jazyka

- ➜ víceznačnost
- ➜ anaforické výrazy
- ➜ indexické výrazy
- ➜ nejasnost
- ➜ nekompozicionalita
- ➜ struktura promluvy
- ➜ metonymie
- ➜ metafory

Víceznačnost

- ➜ *ambiguity*
- ➜ víceznačnost může být lexikální, syntaktická, sémantická a referenční
- ➜ lexikální – "stát," "žena," "hnát"
- ➜ syntaktická – "Jím špagety s masem."

  "Jím špagety se salátem."

  "Jím špagety s použitím vidličky."

  "Jím špagety se sebezapřením."

  "Jím špagety s přítelem."
- ➜ sémantická – "Jeřáb je vysoký." "Viděli jsme veliké oko."
- ➜ referenční – "Oni přišli pozdě." "Můžeš mi půjčit knihu?" "Ředitel vyhodil dělníka, protože (on) byl agresivní."

Anaforické a indexické výrazy

anaforické výrazy:

- ➜ *anaphora*
- ➜ používají zájmena pro odkazování na objekty zmíněné dříve
- ➜ "Poté co se Honza s Marií rozhodli se vzít, (oni) vyhledali kněze, aby je oddal."
- ➜ "Marie uviděla ve výloze prstýnek a požádala Honzu, aby jí ho koupil."

indexické výrazy:

- ➜ *indexicals*
- ➜ odkazují se na údaje v jiných částech promluvy
- ➜ "Já jsem tady."
- ➜ "Proč jsi to udělal?"

Metafora a metonymie

metafora:

- *metaphor*
- použití slov v přeneseném významu (na základě podobnosti), často systematicky
- "Zkoušel jsem ten proces zabít, ale nešlo to."
- "Bouře se vzteká."

metonymie:

- *metonymy*
- používání jména jedné věci pro (často zkrácené) označení věci jiné
- "Čtu Shakespeara."
- "Chrysler oznámil rekordní zisk."
- "Ten pstruh na másle u stolu 3 chce další pivo."

Nekompozicionalita

- *noncompositionality*
- příklady porušení pravidla kompozicionality u ustálených termínů nebo přednost jiného možného významu při určitých spojeních
- "aligátoří boty," "basketbalové boty," "dětské boty"
- "pata sloupu"
- "červená kniha," "červené pero"
- "bílý trpaslík"
- "dřevěný pes," "umělá tráva"
- "velká molekula"

Syntaktická analýza přirozeného jazyka

- velice rozsáhlé gramatiky (desítky až stovky tisíc pravidel)
- silná víceznačnost – někdy až obrovské množství ($>$milióny) možných syntaktických stromů

  Obehnat Šalounův pomník mistra Jana Husa na pražském Staroměstském náměstí živým plotem z hustých keřů s trny navrhuje občanské sdružení Společnost Jana Jesenia.
- existují efektivní algoritmy pro takové gramatiky

  např. tabulkový analyzátor (*chart parser*), beží v $O(n^3)$, tisíce slov/sekundu

Příklad stromu analýzy v systému synt