Úvod do umělé inteligence, jazyk Prolog

Co je umělá inteligence?

- systém, který se chová jako člověk
  - Turingův test
  - 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
- systém, který myslí rozumně
  - např. studia logiky
  - problém - umět najít řešení teoreticky x prakticky (složitost a vyčíslitelnost)
  - problém - neúplnost a nejistota vstupních dat
- systém který se chová rozumně: inteligentní agent - systém, který:
  - jedná s nějakým účelem, jedná samostatně, jedná na základě vstupů ze svého prostředí, pracuje delší dobu, adaptuje se na změny

Stručné shrnutí prologu

- deklarativnost (specifikace programu je přímo programem) - určuje, co bude výstupem programu
- řešení problémů týkajících se objektů a vztahů mezi nimi

Principy:

- backtracking řízený unifikací, hojně využívá rekurzi
  - backtracking - standardní metoda prohledávání stavového prostoru do hloubky, zpětná metoda, metoda pokusů a oprav. Jde o algoritmus sloužící k nalezení všech nebo některých řešení při průchodu (prohledávání) stavovým stromem problému, tak, že postupně prochází jednotlivé kandidáty řešení a jakmile zjistí, že kandidát nemůže být platným řešením (nesplnitelný cíl), vrací se k nejbližšímu minulému bodu s alternativní volbou.
  - unifikace - řídící mechanismus, hledání nejobecnějšího unifikátoru dvou termů

Jde o proces, který převádí 2 nebo více termů na společnou instanci pomocí určitých substitucí, 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')**

• rekurze - opakované vnořené volání stejné funkce (podprogramu), rekurzivní funkce je taková funkce, která při svém běhu volá sama sebe

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

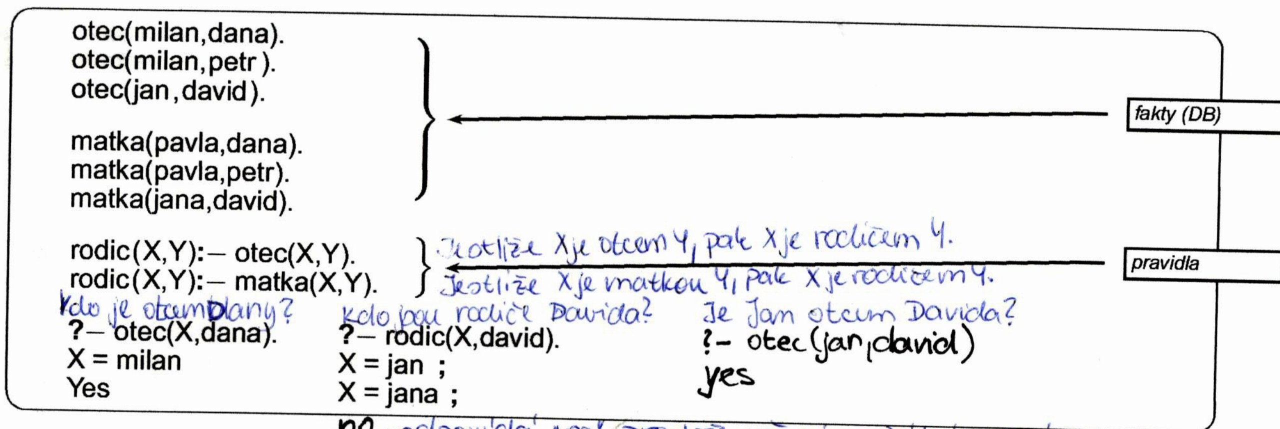
- spojitost s logikou: snaha dokázat pravdivost daného cíle. Cíl je dokázán unifikuje-li s hlavou nějaké klauzule a všechny podcíle v těle této klauzule jsou rovněž dokázány. Strategie výběru podcíle: shora dolů, zleva doprava.

Syntax jazyka prolog

parent(tom, bob)

= Tom je rodičem (parent) Boba; "parent" je jméno vztahu, "tom" a "bob" jsou argumenty vztahu

jednoduchý příklad – DB rodinných vztahů:



no - odpovídá "no" protože už nejsou žádná možná řešení

• logický prologovský program - seznam klauzulí (pravidel a faktů)

• 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: p(Z,X) :- p(X,Y), p(Y,Z)
  - cíl: tělo bez hlavy ?- p(g,f) (dotaz)

Fakty jsou vždy pravdivé, zatímco pravidla jsou pravdivá pokud je splněna nějaká podmínka.

- predikát - seznam všech klauzulí se stejným funktorem (to před závorkou) a aritou v hlavovém literálu (arita - počet termů v závorce). Zapisují se ve tvaru "funktor/arita" - potomek/2
- literál - atomická formule, nebo její negace
- atomická formule - v prologu zcela odpovídá složenému termu (syntaktický rozdíl neexistuje)
  - konstanta - začínají vždy malým písmenem (mohou to být řetězce písmen, čísla i speciálních znaků - a, 1, ':', [], sc2, tomas, anna, ...) a také řetězce v uvozovkách
  - proměnná - začínají velkým písmenem a také podtržítkem (nezajímavá vrácená hodnota), např. X, Vys, _, _A, ...
  - složený term - f(a,X) datové objekty v Prologu: datum(1, kveten, 1983)
    - funktor: datum
    - argumenty: 1, kveten, 1983 (arg/3 - n-tý - tedy 3. - argument)
    - arita: počet argumentů (3)
    - termy jsou unifikovatelné, jestliže jsou identické anebo jestliže proměnné mohou být instanciovány tak, že termy jsou po substituci identické (viz unifikace)
- dotaz - uživatel se ptá programu, zda je něco pravdivé
  - yes: pozitivní odpověď - dotaz je splnitelný a uspěl
  - no: dotaz je nesplnitelný a neuspěl (negativní)
  - prolog umí generovat více řešení pokud existují
- všechny strukturované objekty v Prologu jsou stromy

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

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

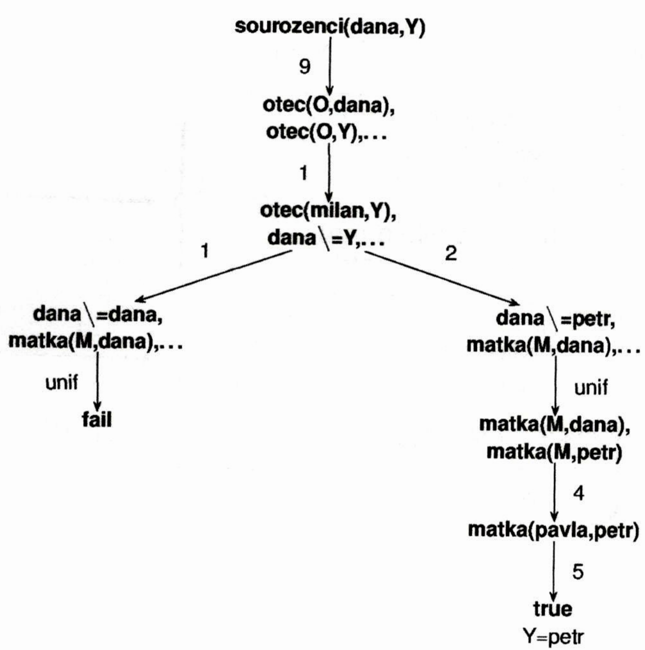
```
1  otec(milan,dana).
2  otec(milan,petr).
3  otec(jan,david).
4  matka(pavla,dana).
5  matka(pavla,petr).
6  matka(jana,david).
7  rodic(X,Y):– otec(X,Y).
8  rodic(X,Y):– matka(X,Y).
```

```
?– 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, dana \= petr          % true
4, matka(M,dana)         % M = pavla
5, 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,
10                    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

```
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.
```

- nekonečná posloupnost přirozených čísel, začínající 0, 1, 1, 2, 3, 5, 8, 13, 21, ..., kde každé číslo je součtem dvou předchozích, rekurentní vzorec této řady je:

$$F(n) = \begin{cases} 0, & \text{pro } n = 0 \\ 1, & \text{pro } n = 1 \\ F(n-1) + F(n-2) & \text{pro } n \geq 2 \end{cases} \quad / \quad \begin{aligned} fib_0 &= 0 \\ fib_1 &= 1 \\ fib_i &= fib_{i-1} + fib_{i-2}, \text{ pro } i \geq 2 \end{aligned}$$

- program výše - exponenciální časová složitost (konstantní 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)).
```

přidá klauzuli v závorce do databáze, vloží ji na začátek databáze

Složitost algoritmů

- časová složitost: kolik času potřebuje program na své vykonání
- paměťová / prostorová složitost: kolik paměti (úložného prostoru) program ke svému běhu potřebuje
- složitost problému - složitost optimálního algoritmu korektně řešícího zadaný problém
- efektivní algoritmy - pracující s lepší než exponenciální / faktoriálovou složitostí

- základní složitosti:
  - konstantní: $O(1) = c$
  - logaritmická $O(\log n)$
  - lineární $O(n)$
  - lineárnělogaritmická $O(n \cdot \log n)$
  - kvadratická $O(n^2)$
  - kubická $O(n^3)$
  - obecná polynomiální $O(n^c)$
  - exponenciální $O(c^n)$
  - faktoriálová $O(n!)$

Asymptotická složitost: určuje náročnost algoritmu tak, že zjišťuje, jakým způsobem se bude chování algoritmu měnit v souvislosti na změně velikosti (počtu) vstupních dat. Zapisuje se pomocí Omikron notace, kde n je veličina popisující velikost vstupních dat.

- např. časová složitost $O(n)$ říká, že doba práce algoritmu se zvýší přibližně tolikrát, kolikrát se zvýší velikost vstupu; u složitosti $O(n^2)$ se doba trvání zvyšuje kvadraticky – pokud se délka vstupu zvýší dvakrát, potřebný čas se zvýší 4x, u složitosti $O(1)$ na vstupu nezáleží, čas je stále stejný (podobně prostorová)

Operace na datových strukturách

Práce se seznamy

Seznam:

- rekurzivní datová struktura – každá položka obsahuje odkaz na položku stejného typu
- uspořádaná posloupnost prvků (libovolných termů včetně seznamů)
- seznam: [a, b, c], prázdný seznam []
- funktor ".", dva argumenty (operátor ./2) – 1. argument je hlava (hlavou je vždy ~~prvek~~ prvek seznamu nebo více prvků), 2. je tělo (seznam zbývajících prvků) → .(Hlava, Tělo) nebo [Hlava | Tělo]

.(a, [])   [a]   [a | []]

.(a, .(b, .(c, [])))   [a, b, c]   [a, b | [c]]   [a | [b, c]]   [a, b, c | []]

[a | [b, c | []]]   [a | [b | [c | []]]]

Prvek seznamu: member(+Prvek,+Seznam)

- je pravdivé pokud v seznamu existuje daný prvek [[a,b],c]
- platí: member(b,[a,b,c]), neplatí: member(b,[[a,b]|[c]])
- X je prvek seznamu S, když
  - X je hlava seznamu S member(X,[X|_])
  - nebo X je prvek těla seznamu S

member(X,[_|Telo]):- member(X,Telo)

- např. ?- member(a,[X,b,c])

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]).
    No
```
```
?– member(a,[a,b,a]),write(ok),nl,fail.
    ok
    ok
    No
```

test na identitu (porovnání bez přiřazení). Jestliže X je identické s Y pak X je hlavou seznamu. nl – přesune se na další řádek (new line). ?- X==1 no; ?- 1==1 yes

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
```

Je-li X identické s Y, pak X je hlavou seznamu. Není-li X identické s Y, pak je prvkem těla seznamu.

fail – vestavěný predikát, nutí program se vracet a vypsat tak všechna řešení

Smazání prvku seznamu del(+A,+L,-VysL)

- smaže všechny výskyty prvku A v seznamu L; -VysL - seznam, kde je daný prvek smazaný

del1(+A,+L,-VysL) smaže vždy jeden (podle pořadí) výskyt prvku A v seznamu L - je nedeterministické, tudíž je-li tam vícekrát daný prvek, může smazat kterýkoliv z nich pomocí backtrackingu

```
del(_,[],[]).
del(A,[A|T],V) :– del(A,T,V).
del(A,[H|T1],[H|T2]) :– A\=H, del(A,T1,T2).

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

pokud smažu lib. prvek z prázd. seznamu, výstup je prázd. sez.; smažu A ze seznamu a výstup jde do prom. V; vymažu A z těla T a uložím do V

- jestliže X je hlavou seznamu S pak výsledkem je tělo S; jestliže X je v těle

seznamu, pak X je označán až v těle

Rozšíření seznamu o daný prvek insert(+A, +L, -VysL)

- postupně vkládá (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
```

A přidám k seznamu L, výstupem je nový seznam, který má hlavu A

A přidám do T1 a vlož. do T2

Permutace

- permutace z n prvků je každá n-členná variace z těchto prvků, je uspořádaná tak, že každý z prvků se vyskytuje právě jednou, u variace záleží na pořadí; ze zadaných prvků postupně vybereme všechny, každá permutace tedy odpovídá nějakému pořadí zadaných prvků
- výpočet: $P(n) = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 = n!$
- 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
```

23 vlož do V pak to prohledám

L je permutací [H|T], přičemž do permutace těla seznamu vkládá insert postupně na všechny pozice hlavu pův. seznamu ⇒ vznik permutovaného seznamu L

Argumenty jsou 2 listy, kde jeden je permutací druhého.

No - není další řešení

- pomocí del1

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

- smaže prvek X z původního seznamu, provede permutaci zbytku tohoto seznamu a pak přidá X před takto vytvořený seznam P
- pomocí append

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

Spojení seznamů: append (?Seznam1, ?Seznam2, ?Seznam)

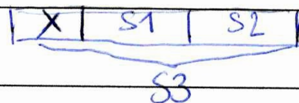
- Seznam je spojení seznamů Seznam1 a Seznam2
- platí: append ([a,b], [c,d], [a,b,c,d])
- neplatí: append ([b,a], [c,d], [a,b,c,d])
  append([a,[b]], [c,d], [a,b,c,d])
- pokud je 1. argument prázdný seznam, pak 2. a 3. argument jsou stejné seznamy

append([],S,S)

- pokud je 1. argument neprázdný seznam, pak má 3. argument stejnou hlavu jako 1.

append([X|S1], S2, [X|S3]) :- append (S1, S2, S3)

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



predikát **append** je **vícesměrný**:

```
?– append([a,b],[c,d],L).          - spojení seznamů
   L = [a, b, c, d]
   Yes
?– append(X,[c,d],[a,b,c,d]).      - vyhledávání v seznamu
   X = [a, b]
   Yes
?– append(X,Y,[a,b,c]).            - dekompozice seznamu na dva seznamy
   X = []          Y = [a, b, c];
   X = [a]         Y = [b, c];
   X = [a, b]      Y = [c];
   X = [a, b, c]   Y = [];
   No  - možná řešení jsou vyčerpána
```

predikát **append** je všestranně použitelný:

```
member(X,Ys)          :– append(As,[X|Xs],Ys).  X je prvkem Ys, pokud Ys můžeme rozdělit na 2 seznamy, přičemž hlavou 2. je X
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).
(sousedící, přilehlý)
```

- rozdílové seznamy (difference lists) - efektivita řešení predikátu append, zapisují se 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**

- Seznam 2 jako volná proměnná slouží jako "ukazatel" na konec Seznamu 1
- append_dl - predikát append s rozdílovými seznamy.

```
append_dl(A−B,B−C,A−C).
```
A − B B − C A−C ([a,b|X] − Y) X = B = [c,d|Y] =>
=> A = [a,b, c,d | Y]

```
?− 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): qsort(+L, -VysL)

– třídí seznam L technikou rozděl a panuj

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

dělení prvků na hlavu a tělo (oddělení těla) — L=[H|T],H=5

T=[3,7,8,1,4,7,6]

divide(5, . . .)

$\forall$prvky ≤ 5 — M=[3,1,4], qsort(M)

rozdělení prvků na seznamy prvků menších než hlava a větších než hlava

$\forall$prvky > 5 — V=[7,8,7,6], qsort(V)

M1=[1,3,4] — seřazení obou seznamů — V1=[6,7,7,8]

append - M1.[5].V1

Vysl=[1,3,4,5,6,7,7,8] — složení seznamů menších a větších + hlavy

```
qsort ([],[])   :− !.        % “ řez ” − zahoď další možnosti řešení
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).
```

Poznámky u kódu: divide – rozdělí seznam na hlavu, tělo, prvky menší a větší než hlava; qsort – seřadí seznamy menších a větších prvků; append – složí seznamy prvků menších než hlava a větších než hlava (kam pův. hlavu přidá jako novou hlavu) => seznam seřazených prvků

! = řez, je to vestavěný predikát, který se používá pokud chceme programu či uživateli zakázat, aby používal navracení. Často se používá u predikátů, které mají mít nejvýše jedno řešení

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),      A2−Z2
                      qsort_dl(L1,A−[H|A1]).  −Z1
                                              A1−[X|A2]
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).
```

L1 – menší, L2 – větší
Seznam seřazených menších A−Z1
Seznam seřazených větších A1−B
Spojení menších a [H|větších] koresponduje se spojením A−Z1 a [H|A1]−B
Výsledný seznam je reprezentován A−B, kde Z1=[H|A1]

[H|A1]−Z1

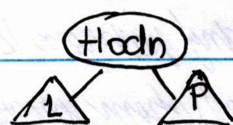
Binární stromy

Uspořádané binární stromy:

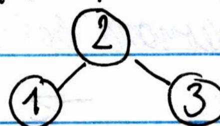
- orientovaný graf s jedním vrcholem (kořenem), z něhož existuje cesta do všech vrcholů grafu. Každý vrchol mimo kořen má 1 předka a 2 syny.
- u uspořádaného binárního stromu jsou všichni přímí potomci každého uzlu seřazeni. Pokud má uzel nelist, lze určit prvního přímého potomka, druhého přímého potomka až n-tého přímého potomka
- reprezentace binárního stromu

nil prázdný strom   t(L, Hodn, P) strom

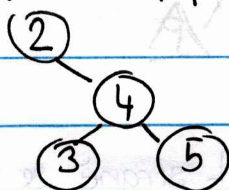
levý podstrom (strana)   vrchol   pravý podstrom (pravá strana)



, např.   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

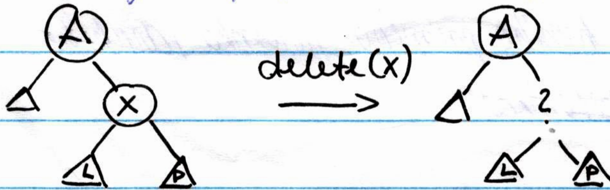
```
addleaf(nil,X,t( nil ,X, nil )).
addleaf(t(Left,X,Right),X,t(Left,X,Right)).
addleaf(t(Left,Root,Right),X,t(Left1,Root,Right)) :- Root>X,addleaf(Left,X,Left1).
addleaf(t(Left,Root,Right),X,t(Left,Root,Right1)) :- Root<X,addleaf(Right,X,Right1).

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

- výsledkem přidání X do prázdného stromu T je strom t(nil, X, nil)
- jestliže X je kořenem daného stromu, pak se rovnají, žádný prvek - duplikát se nevkládá

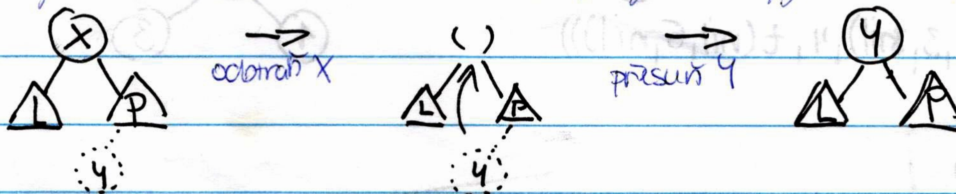
- jestliže kořen stromu je větší než X, pak X vložíme do levého podstromu; jestliže kořen stromu je menší než X, pak X vložíme do pravého podstromu

Odebírání z binárního stromu

- predikát addleaf není všesměrný, proto nelze definovat del(T, X, T1) :- addleaf(T1, X, T) - inverzní operaci k přidání bin. stromu. Pokud smažeme X z vnitřního vrcholu, bude ve stromu díra, L a P nebudou spojeny se zbytkem stromu, nelze je ani obě napojit na A (zbyla by jen jedna strana)



- opačný postup:
  - pokud je odebíraná hodnota v listu, nahradí se hodnotou nil
  - pokud jeden z podstromů L a P je prázdný, přiřadíme neprázdný podstrom k A
  - pokud je odebíraná hodnota v kořeni (pod)stromu, je nutné (pod)strom přestavět



Uzel (List) nejvíce vlevo na P straně / nejvíce vpravo na L straně se přesune do pozice kořene

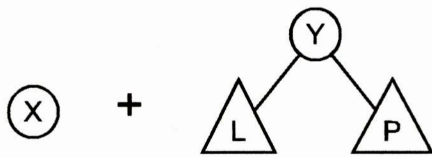
- delleaf(+T, +X, -Vysl) odstraní ze stromu T uzel s hodnotou X

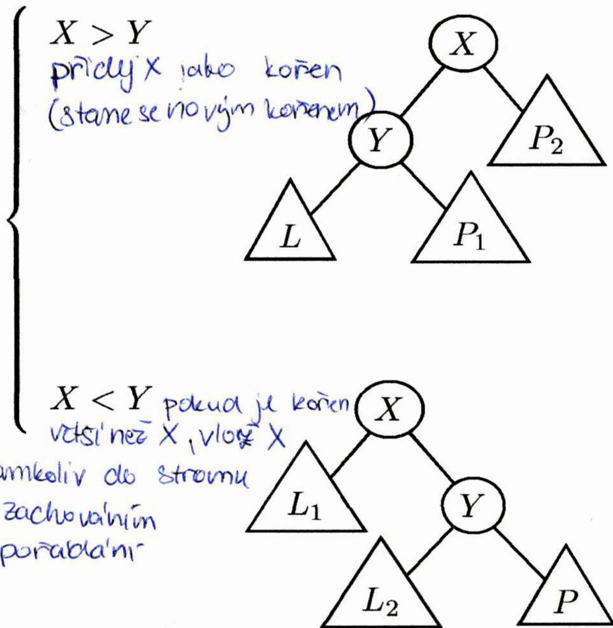
```
delleaf(t(nil,X,Right),X,Right).
delleaf(t(Left,X,nil),X,Left).
delleaf(t(Left,X,Right),X,t(Left,Y,Right1)) :- delmin(Right,Y,Right1).
delleaf(t(Left,Root,Right),X,t(Left1,Root,Right)) :- X<Root,delleaf(Left,X,Left1).
delleaf(t(Left,Root,Right),X,t(Left,Root,Right1)) :- X>Root,delleaf(Right,X,Right1).

delmin(t(nil,Y,R),Y,R).
delmin(t(Left,Root,Right),Y,t(Left1,Root,Right)) :- delmin(Left,Y,Left1).
```

- delmin - přesouvá uzel nejvíce vpravo/vlevo v levém/pravém podstromu
  - Y je minimální (nejvíc vlevo/vpravo) uzel stromu
  - Tree1 (Right1/Left1) je strom, odkud je Y smazáno

Všesměrný algoritmus pro vkládání/odebírání add(?T, +X, ?Vysl)

- přidá do binárního stromu T uzel s hodnotou X s přeuspořádáním stromu (jako kořen nebo jinam při navracení)






```
% přidej jako kořen                                  = nový kořen
add(T,X,T1) :- addroot(T,X,T1).
% nebo kamkoliv do stromu (se zachováním uspořádání)
add(t(L,Y,R),X,t(L1,Y,R)) :- gt(Y,X),add(L,X,L1).   vlož X do levého podstromu
add(t(L,Y,R),X,t(L,Y,R1)) :- gt(X,Y),add(R,X,R1).   vlož X do pravého podstromu
addroot(nil,X,t( nil ,X, nil )).                     vlož X do prázdného stromu
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)).
```

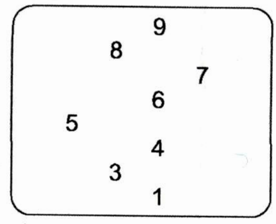
- soubor uzlů v L1 a L2 odpovídá souboru uzlů v L, L1 obsahuje uzly menší než X, L2 pak větší než X (to samé platí pro P1 a P2)
- definice gt(X,Y) = greater than; na konkrétním uživateli
- funguje i "obráceně" => lze definovat del(T,X,T1) :- add(T1,X,T)

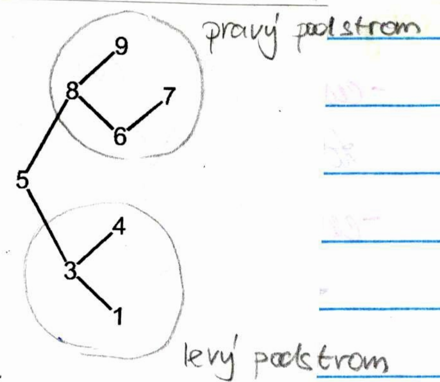
Výpis binárního stromu:

- 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 )))
```

→





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

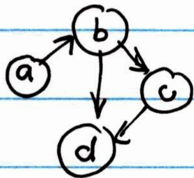
```
show(T) :- show2(T,0).          zobraz binární strom
show2(nil,_).                   zobraz strom odsazený pomocí Indent
show2(t(L,X,R),Indent) :- Ind2 is Indent+2,show2(R,Ind2),tab(Indent),
    write(X),nl,show2(L,Ind2).
```

(poznámky: write(X) – zapiš kořen; Ind2 is Indent+2 – odsazení podstromů; show2(R,Ind2) – zobraz pravý podstrom; show2(L,Ind2) – zobraz levý podstrom)

Reprezentace grafů

- graf je definován souborem vrcholů a hran (páry vrcholů)
- způsoby reprezentace v Prologu:

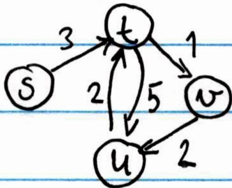
① term graph(V,E), kde V je seznam vrcholů a E je seznam hran grafu. Každá hrana je ve tvaru e(V1,V2), kde V1 a V2 jsou vrcholy grafu



G = graph([a,b,c,d],[e(a,b),e(b,d),e(b,c),e(c,d)])

znázorňuje **orientovaný graf**

② vgraph(V,E) definuje uspořádanou dvojici seznamů vrcholů (V) a hran (E). Hrany jsou ve tvaru a(PocatecniV, KoncovyV, CenaHrany)

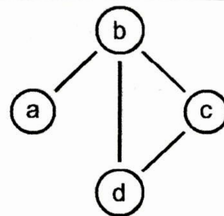


G = vgraph([s,t,u,v],[a(s,t,3), a(t,v,1), a(t,u,5), a(u,t,2), a(v,u,2)])

znázorňuje **orientovaný ohodnocený graf**

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

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



díky přidanému pravidlu představuje **neorientovaný graf** (bez pravidla je orientovaný). V Prologu nelze přímo zapsat neorientovaný graf.

Cesty v grafech:

- cesta nesmí obsahovat žádný cyklus => vrchol se může objevit jen 1x, je to posloupnost vrcholů a hran
- cesta v neorientovaném grafu: path(+A, +Z, +Graf, -Cesta)
  - v grafu Graf najde z vrcholu A do vrcholu Z cestu Cesta (Graf je ve tvaru 1)
  - Cesta je reprezentována jako seznam vrcholů cesty, např. path(a,d,G,[a,b,d])
  - jestliže A = Z pak Cesta = [A], jinak hledáme acyklickou cestu path1

z nějakého vrcholu Y do Z a poté cestu z A do Y vyhýbajíce se vrcholům path1

```
path(A,Z,Graf,Cesta) :- path1(A,[Z],Graf,Cesta).      vztah mezi path a path1

path1(A,[A|Cesta1],_,[A|Cesta1]).       sousedící           podmínka zamezující cyklu
path1(A,[Y|Cesta1],Graf,Cesta) :- adjacent(X,Y,Graf),not(member(X,Cesta1)),
    path1(A,[X,Y|Cesta1],Graf,Cesta).
                                  hrany X-Y a Y-X jsou členem množiny hran
adjacent(X,Y,graph(Nodes,Edges)) :- member(e(X,Y),Edges);member(e(Y,X),Edges).
```

- path1(A, Cesta1, Graf, Cesta) - A je vrchol, Graf je graf, ~~Pat~~ Cesta1 je cestou v grafu, Cesta je acyklická cesta, která jde z vrcholu A na začátek Cesty1 a pokračuje Cestou1 až k jejímu konci
- pokud startovací vrchol Cesty1 (Y) nesplývá se startovacím vrcholem Cesty (A), hledáme vrchol X, který sousedí s vrcholem Y, není součástí Cesty1 a naplňuje vztah path1(A, [X| Path1], Graf, Cesta)
- adjacent (X, Y, Graf) - znamená, že mezi X a Y je v grafu Graf spojnice
- 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

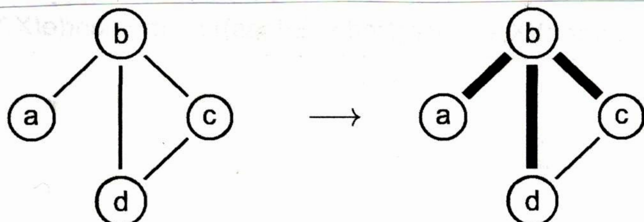
```
                                          cena Cesty 1
path(A,Z,Graf,Cesta,Cena) :- path1(A,[Z],0,Graf,Cesta,Cena).
                                           Cesta1
path1(A,[A|Cesta1],Cena1,Graf,[A|Cesta1],Cena1).          cena hrany
path1(A,[Y|Cesta1],Cena1,Graf,Cesta,Cena) :- adjacent(X,Y,CenaXY,Graf),
    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).
```

~~grafem~~ ~~je~~ Graf je seznam hran ve tvaru X-Y/CenaXY (viz adjacent)
- cena cesty je součtem cen hran dané cesty

Kostra grafu: strom, který prochází všechny vrcholy grafu a jehož hrany jsou zároveň hranami grafu, neobsahuje žádný cyklus a je souvislý (spanning tree)

stree(Graph, Tree) Tree je kostrou grafu Graph

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

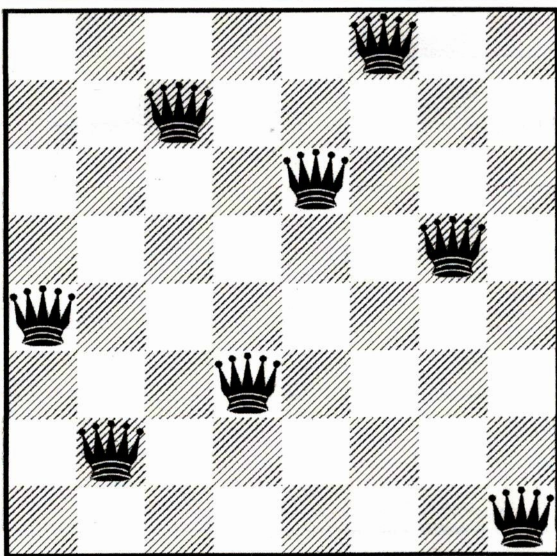


```
stree(Graph,Tree) :- member(Edge,Graph),spread([Edge],Tree,Graph).

spread(Tree1,Tree,Graph) :- addedge(Tree1,Tree2,Graph),spread(Tree2,Tree,Graph).
spread(Tree,Tree,Graph) :- not(addedge(Tree,_,Graph)).
                            % žádnou hranu nemůžeme přidat, aniž bychom nevytvořili cyklus

addedge(Tree,[A-B|Tree],Graph) :- adjacent(A,B,Graph),node(A,Tree),  % A je v Tree
                                   % vrcholy A a B spolu sousedí v Graph
    not(node(B,Tree)).
    % A-B nevytváří v Tree cyklus
adjacent(A,B,Graph) :- member(A-B,Graph);member(B-A,Graph).

node(A,Graph) :- adjacent(A,_,Graph).  % A je vrcholem v Graph jestliže A sousedí s čímkoliv v Graph
```

- spread(Tree1,Tree,Graph) - Tree1 se "rozšíří" do kostry grafu Tree
- addedge(Tree,NewTree,Graph) - přidá hranu z Graph do stromu Tree bez vytvoření cyklu

Prohledávání stavového prostoru

Problém osmi dam



Úkol: Rozestavte po šachovnici 8 dam tak, aby se žádné 2 vzájemně neohrožovaly

- nesmí stát v 1 řádku, v 1 sloupci ani v 1 diagonále
- výsledkem je seznam 8 souřadnic
- celkem pro 8 dam existuje 92 různých řešení (nezáleží na pořadí, některé tedy vizuálně vypadají stejně)

Reprezentace pozic

- datová struktura - osmiprvkový seznam [X1/Y1, X2/Y2, X3/Y3, X4/Y4, X5/Y5, X6/Y6, X7/Y7, X8/Y8], každý prvek koresponduje s jednou dámou a specifikuje její pozici
- protože víme, že všechny dámy musí musí být v rozdílných sloupcích, aby se vyhly vertikálnímu útoku, můžeme souřadnice X dát fixní:

[1/Y1, 2/Y2, 3/Y3 ... 8/Y8]

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

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

sol ([]).          % nemám žádnou dámu, která by ohrožovala
sol([X/Y|Others]) :- sol(Others),
                     member(X,[1,2,3,4,5,6,7,8]),      % umístím zbývající dámy
                     member(Y,[1,2,3,4,5,6,7,8]),
                     noattack(X/Y,Others).          % otestuji jestli se pozohrožují

noattack(_ ,[]).
noattack(X/Y,[X1/Y1|Others]) :- X=\=X1, Y=\=Y1, Y1-Y=\=X1-X, Y1-Y=\=X-X1,
                                noattack(X/Y,Others).
        % rozdílné souřadnice; rozdílené diagonály

template([X1/Y1, X2/Y2, X3/Y3, X4/Y4, X5/Y5, X6/Y6, X7/Y7, X8/Y8]).   % fakt, seznam 8 dam rekurzivním způsobem

% zavolá se to volnou proměnnou
?- 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
```

- [X/Y|Others] - první dáma je na nějakém poli X/Y a ostatní jsou na polích specifikovaných pomocí seznamu Others
  - v Others nesmí nastat útok, tzn. Others musí být samotné také řešením
  - X/Y musí být celá čísla mezi 1 a 8
  - dáma na poli X/Y nesmí napadat žádnou dámu z Others
- noattack (X/Y, [X1/Y1|Others])
  - královna na X/Y nesmí napadat královnu na X1/Y1
  - královna na X/Y nesmí napadat žádnou z královen v Others, tzn.
  - souřadnice Y i X jsou odlišné
  - nejsou ve stejné diagonále → vzdálenost mezi polem v X-direction se nesmí rovnat vzdálenosti v Y-direction
- počet možných řešení $|| = 64 \cdot 63 \cdot 62 \cdots 57 \approx 1{,}8 \times 10^{14}$ (pro 8 dámu počet možných řešení)

=> omezení stavového prostoru - každá dáma má svůj sloupec, další dáma nemůže být ve stejném sloupci, počet možných řešení $|| = 8 \cdot 7 \cdot 6 \cdots 1 = 40320$

```
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]).
```

Do template jsou zakódovány pevné souřadnice sloupců => zjednodušení.

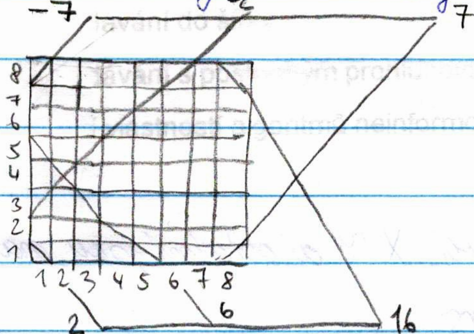
Problém n dam pro $n = 100$:

řešení I.... 10^{400} řešení II.... 10^{158} řešení III.... 10^{52}

- přidání souřadnic diagonál – k souřadnicím x a y přidáme souřadnice diagonál u a v – pootočíme šachovnici o 45°, souřadnice x a y pak popisují souřadnice diagonál u, v

$u = x - y$ $D_x = [1..8]$ → $D_u = [-7..7]$

$v = x + y$ $D_y = [1..8]$ $D_v = [2..16]$



Řešení: vyber 8 čtveřic (X, Y, U, V) z domén (D_x, D_y, D_u, D_v) a nikdy nepoužij stejný prvek dvakrát. Vybereme pozici první dámy, smažeme dané prvky ze 4 domén a zbytek použijeme k umístění zbylých dam.

Po každém umístění dámy aktualizujeme domény volných pozic → počet možných řešení III = 2 057.

```
solution(YList) :– sol(YList ,[1,2,3,4,5,6,7,8],[1,2,3,4,5,6,7,8],      (Dx, Dy)
                      [–7,–6,–5,–4,–3,–2,–1,0,1,2,3,4,5,6,7],              diagonály Du ↗
                      [2,3,4,5,6,7,8,9,10,11,12,13,14,15,16]).             diagonály Dv ↘
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).
                                                   (používá zbývající hodnoty)
% když del nenajde Item, končí neúspěchem
del(Item,[Item| List ], List ).
del(Item,[ First | List ],[ First | List1 ]) :– del(Item, List , List1 ).
```

- sol(YList) – Y souřadnice dam
- del(Y, Dy, Dy1) – vybere souřadnici Y
- U is X−Y – korespondující diagonála zdola nahoru ↗ (u)
- del(U, Du, Du1) – smaže diagonálu u
- V is X+Y – korespondující diagonála shora dolů ↘ (v)
- del(V, Dv, Dv1) – smaže diagonálu v

Prohledávání stavového prostoru

- od určitých velikostí n je těžké hledat jiné řešení (velká složitost) ⇒ prohledávání stavového prostoru

Řešení problému prohledáváním stavového prostoru

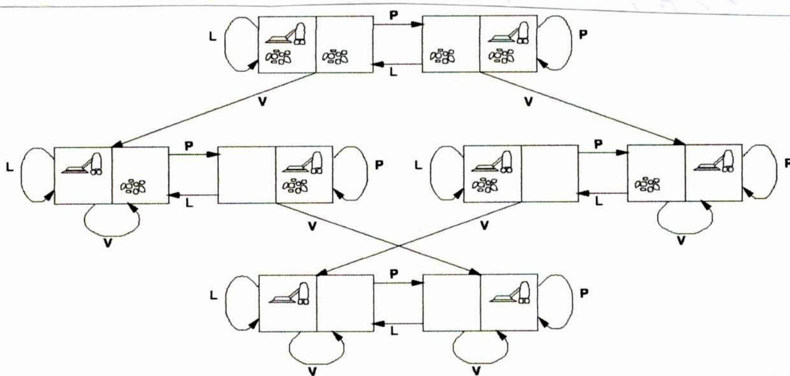
- předpoklady – statické a deterministické prostředí (nelišíme maximální stavy vždy se nacházíme pouze v 1 stavu. Statické prostředí – mění se jen po

okolní z vnějšku (agenta), deterministické pr. - ze znalosti momentálního stavu prostředí a vykonávané akce lze predikovat následný stav.

předpoklady - statické a deterministické prostředí, diskrétní stavy

- **stavový prostor** - řešenou doménu si rozdělíme do různých stavů
- **počáteční stav** - init(State)
- **cílová podmínka** (stav) - goal(State) vrací True/False - podle toho, zda daný stav je řešením nebo ne
- **přechodové akce** move(State, New State) generuje všechny sousedící stavy z libovolného stavu, opakovaným voláním move vyčerpáme celý stavový prostor
- **prohledávací strategie** - prohledávací strom = pořadí v jakém jsou prohledávány jednotlivé uzly (uzel prohledávacího stromu: stav, rodičovský uzel, přechodová akce, hloubka uzlu, cena - g(n) cesty, c(x, a, y) přechodu)

Problém agenta vysavače



- máme 2 místnosti (L, P)
- jeden vysavač buď v L nebo P místnosti - 2 možnosti
- v každé místnosti je/není špína - 2 možnosti

$\Rightarrow$ počet stavů $2 \times 2^2 = 8$

šipky značí jednotlivé přechody

- akce = {doleva, doprava, Vysávej}
- cíl - vysát všechnu špínu

Abstrakce prohledávání stavového prostoru

- **prohledávací strom** - má kořenový uzel (inicální stav), další uzly jsou generovány funkcí move, zaznamenává náš postup - víme, odkud jsme přišli, jakou přechodovou funkcí, v jaké hloubce je uzel a popř. známe cenu cesty z init. stavu
- **kořenový uzel** - počáteční stav, při prohledávání nejprve testujeme, zda není cílový, pokud ne, provedeme expanzi stavu, čímž generujeme novou množinu stavů k prohledávání
- **uzel prohledávacího stromu**: obsahuje
  - aktuální stav
  - odkaz na rodičovský uzel
  - přechodovou akci, která vedla od rodiče do aktuálního stavu

- hloubku uzlu (počet kroků od kořene)
- cenu cesty z kořene $g(n)$ a přechodu $c(x, a, y)$ (zmá např. nejmenší cenu)

- (optimální) řešení - optimální řešení je nejlepší řešení ze všech, úplné řešení nalezne řešení vždy, když existuje; obecně řešení je cesta z počátečního stavu do cílového
- časová a prostorová složitost

Posunovačka

počáteční stav (např.)

7	2	4
5		6
8	3	1

→ ... →

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 3x3, posunování 8 kamenů
- stavy - pozice všech kamenů
- akce - "pohyb" prázdného místa
- optimální řešení obecné n-posunovačky je NP úplné (deterministicky polynomiální?)
- počet stavů u 8-posunovačky ... $9!/2 = 181\,440$
  u 15-posunovačky 10^{13}
  u 24-posunovačky 10^{25}

Reálné problémy řešitelné prohledáváním - hledání cesty z místa A do B, hledání itineráře, problém obchodního cestujícího (nejkratší možná cesta procházející zadanými body na mapě), návrh VLSI čipu, navigace (auta, robota, ...), postup práce automatické výrobní linky, návrh proteinů (3D sekvence aminokyselin), internetové vyhledávání informací

Řešení problému prohledáváním

- kostra algoritmu: (rekurzivní volání predikát řešící strategie)

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

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

(move – definuje prohledávací strategii)

- porovnání strategií:
  - **úplnost** - Pokud existuje řešení, najde ho?

- optimálnost - najde algoritmus optimální řešení
- časová složitost - Jak dlouho trvá nalezení řešení?
- prostorová složitost - Kolik paměti pro prohledávání potřebujeme?

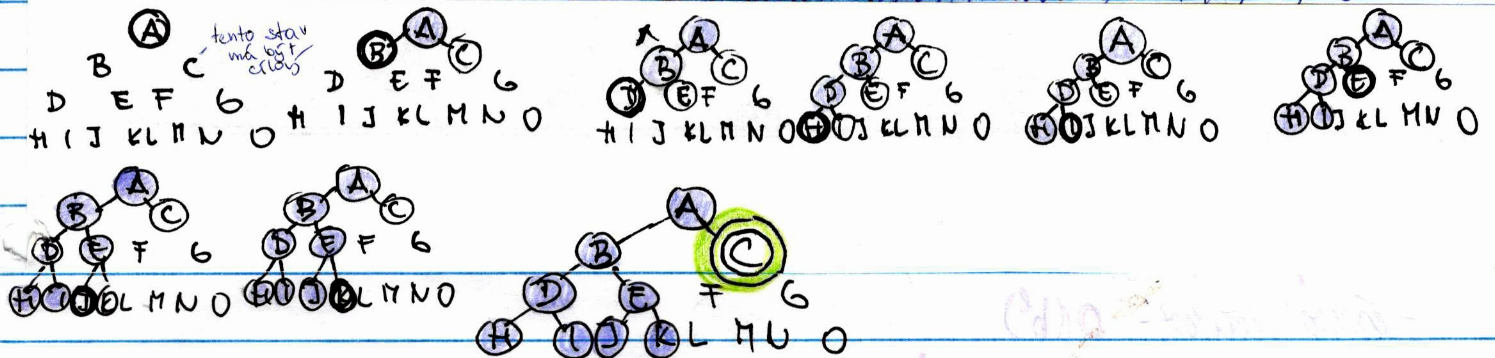
- složitost závisí na:
  - b - faktor větvení (branching factor) - maximální počet následníků
  - d - hloubka cíle (goal depth) popisující proměnná
  - m - maximální hloubka větve/cesty (délka) (maximum depth path), může být ∞?

neinformované prohledávání

- nemá žádnou dodatečnou informaci o stavech (stavovém prostoru), kromě popisu úlohy => umí rozpoznat cílový stav od necílového a generovat následníky, musí systematicky procházet všechny uzly, dokud nenalezne řešení, jsou slepá
- jsou to: prohledávání do hloubky, do hloubky s limitem, do šířky, podle ceny, s postupným prohlubováním

Prohledávání do hloubky

- prohledává se vždy nejlevější a nejhlubší neexpandovaný uzel (Depth-first Search-DFS)



- v procedurálním programovacím jazyce se uzly uloží do zásobníku (fronty LIFO) X Prolog využívá rekurzi

počáteční uzel

```
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).
```

rozšířením cesty [Node|Path] k cíli dostaneme řešení (Solution)

not(member(...)) – zabraňuje vytvoření cyklu

- hledáme cestu řešení z daného bodu (Node) do bodu cílového (goal(Node)).
- dept_first_search (Path, Node, Solution)
  - Node - stav, z nějž vede cesta do cílového stavu
  - Path - cesta, seznam uzlů, mezi počátečním uzlem a Node

• Solution – je ~~cesta~~ Path prodloužená o ~~další uzly~~ do cílového uzlu pomocí Node

Vlastnosti

- **není úplný** – nemusí najít řešení (nekonečná větev, cykly)
- **není optimální** – nemusí najít optimální řešení
- časová složitost – $O(b^m_{max})$, kde m je maximální hloubka a b faktor větvení
- paměťová složitost – $O(bm)$ – lineární, stačí si pamatovat jednu cestu z kořene společně se sousedními uzly, expandovaný uzel s prozkoumanými potomky lze odstranit (m je maximální navštívená hloubka)
- největší problém – nekonečná větev = nenajde se cíl, program neskončí

Prohledávání do hloubky s limitem

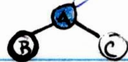
- řešení nekonečné větve – použití "zarážky" = limit hloubky l

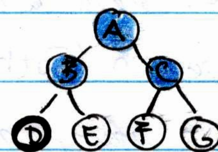
```
solution(Node,Solution) :– depth_first_search_limit(Node,Solution,ℓ).

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).
```

- depth_first_search_limit(Node, Solution, Maxdepth) – prohledávání není dovoleno jít za hodnotu limitu, uzly v hloubce l se považují za uzly bez následníků
- neúspěch (fail) má dvě možné interpretace – vyčerpání limitu nebo neexistenci řešení
- **není úplný** – pokud l (limit) < d (hloubka cíle), řešení nenajdeme
- není **optimální** – pokud l > d, často prohledáváme zbytečně mnoho
- časová složitost – $O(b^l)$
- prostorová složitost – $O(bl)$
- záleží na dobré volbě limitu l – podle znalosti problému

Prohledávání do šířky

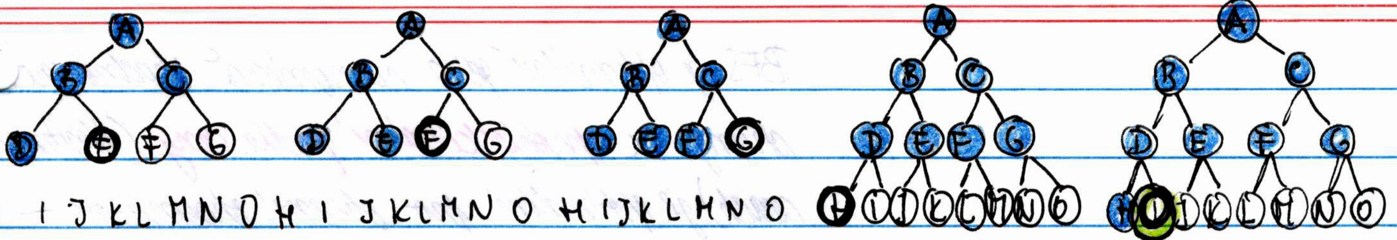
- prohledává se vždy nejlevější uzel s nejmenší hloubkou (Bread-first Search–BFS)
- postupuje se po patrech zleva doprava – z okraje vybereme uzel s nejmenší hloubkou, tedy nejprve expandujeme celou vrstvu stromu





D E F G D E F G D E F G

H I J K L M N O H I J K L M N O H I J K L M N O H I J K L M N O



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

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 $g \leq C^*$, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

```
solution(Start,Solution) :- breadth_first_search([[Start]], Solution).

breadth_first_search([[Node|Path]|_],[Node|Path]) :- goal(Node).
breadth_first_search([[N|Path]|Paths],Solution) :-      % Path: cesta k aktuálnímu uzlu
    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)])])**

- breadth_first_search (Paths, Solution) je pravdivé, pokud některá cesta ze seznamu cest Paths může být rozšířena k cílovému uzlu – taková cesta je řešením Solution.
- seznam cest – každá cesta je seznamem uzlů v opačném pořadí -> nejdříve je generována hlava a posledním prvkem bude počáteční uzel
- bagof – vestavěný predikát, vytvoří seznam -Sezn ze všech termů +Prom, pro něž je splněn cíl +Cíl – generuje všechna jednokroková rozšíření
- řešením je cesta ze Startu do cíle (v opačném pořadí)
- M – nový uzel, N – uzel

Vlastnosti

- je úplný – pro konečné b (za předpokladu konečného větvení)
- je optimální podle délky cesty / není optimální podle obecné ceny
- časová složitost (počet navštívených uzlů pro cíl v hloubce d, který je na konci cesty) – $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$ – exponenciální v d
- prostorová složitost je stejná jako časová (v paměti drží všechny navštívené uzly) – $O(b^{d+1})$
- s hloubkou času potřebujeme informované strategie prohledávání
- největší problém – nároky na paměť, ani čas není dobrý -> potřebujeme informované strategie prohledávání

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

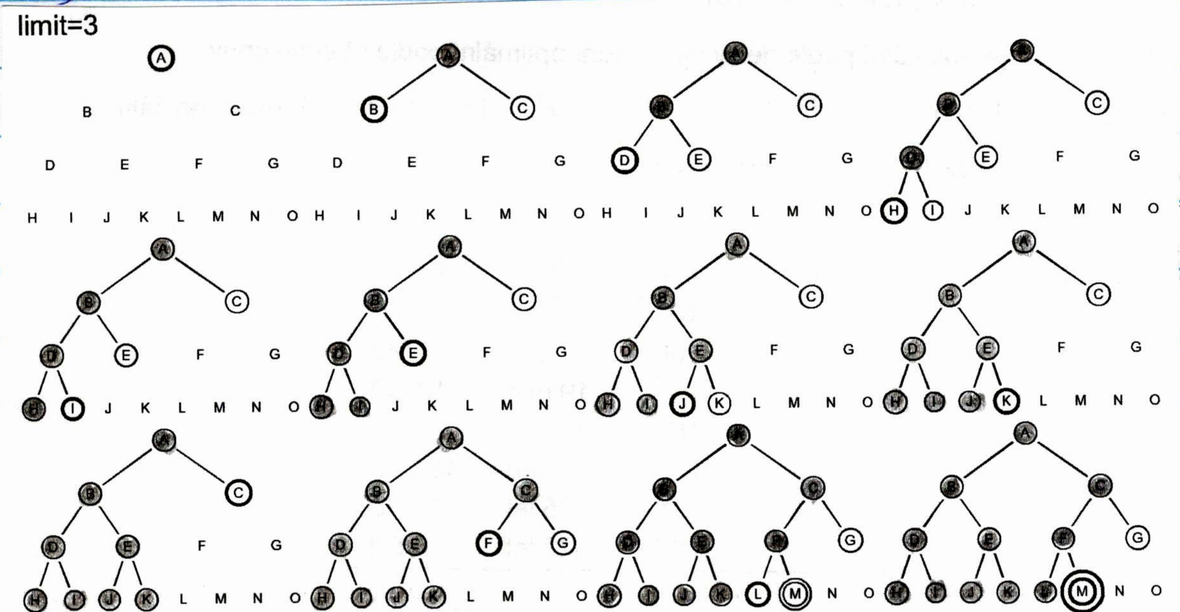
BFS je optimální pro rovnoměrně ohodnocené hrany × prohledávání podle ceny (Uniform-cost search) je optimální pro obecné ohodnocení – frontu uzlů se udržuje uspořádanou podle ceny cesty. => Jako první expandujeme uzel s nejmenší cenou cesty od kořene.

Vlastnosti:

- je úplný, pokud je cena každé akce ≥ konstanta ε
- je optimální pro cena $\geq \varepsilon$, $g(n)$ roste (cena cesty z kořene)
- časová složitost počet uzlů s $g \leq C^*$ (kde C^* je cena optimálního řešení), $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$
- prostorová složitost počet uzlů s $g \leq C^*$, $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$

Prohledávání s postupným prohlubováním

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



- kombinuje výhody BFS a DFS => nízké paměťové nároky (lineární), optimálnost, úplnost
- je úplný pro konečné větvení b
- je optimální (pro cenu $g(n)$ závislou na hloubce (rovnoměrně neklesající funkce hloubky))

• časová složitost $d(b) + (d-1)b^2 + \ldots + 1(b^d) = O(b^d)$
• prostorová složitost $O(bd)$ - lineární

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

$N(IDS) = 50 + 400 + 3000 + 20\,000 + 100\,000 = 123\,450$ prozkoumá uzlů

každé patro prohledá jenom jednou

$N(BFS) = 10 + 100 + 1000 + 10\,000 + 100\,000 + 999\,990 = 1\,111\,100$

- 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	$O(bm)$	$O(b\ell)$	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bd)$

- BFS - garantuje nalezení nejkratšího řešení nejdříve (neplatí pro ohodnocené cesty)
- IDS - také nalezne nejdříve nejkratší řešení

Heuristiky, best first search, A* search

Informované prohledávání stavového prostoru

• Neinformované p. - DFS, BFS a varianty, nemá (téměř) žádné informace o pozici cíle - slepé prohledávání, zná pouze - počáteční/cílový stav, přechodovou funkci
• Informované p. - má navíc informaci o (odhadu) blízkosti stavu k cílovému stavu - heuristická funkce (heuristika) - funkce $h(n)$, která odhaduje délku nejkratší (nejlevnější) cesty do cíle (heureka - objevení něčeho nového)

Heuristické hledání nejlepší cesty

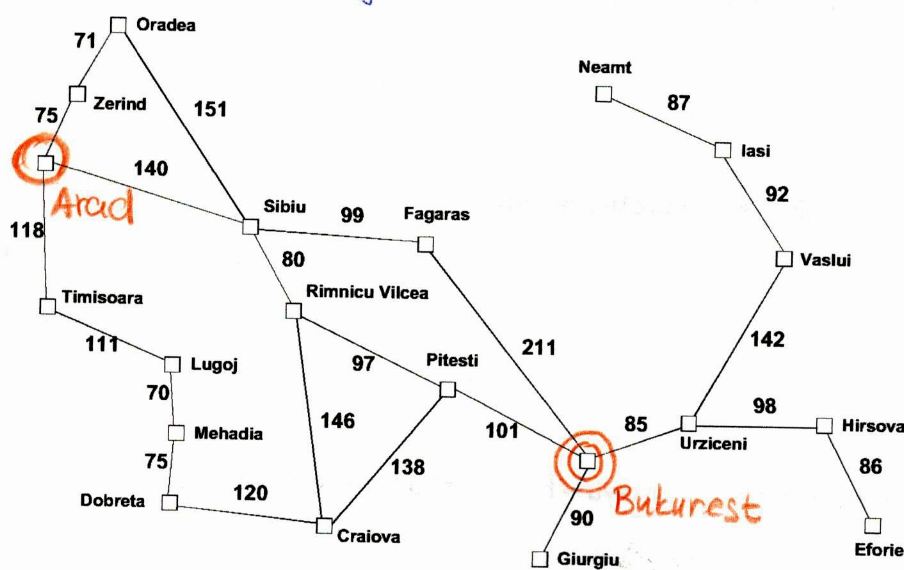
- Best-first Search
- pro každý uzel použijeme ohodnocovací funkci $f(n)$ - obvykle se vybírá uzel o nejmenším ohodnocením, ohodnocení měří vzdálenost k cíli (ale o výpočet přesnou

daného uzlu - vybíráme ten, který se zdá nejpřínosnější)

- udržujeme seznam uzlů uspořádaný (vzestupně) vzhledem k $f(n)$
- pro každý uzel používáme také heuristickou funkci $h(n)$ - slouží k odhadu vzdálenosti daného uzlu od cíle

vždy končím 0 = dorazil do cíle
$n = Goal$

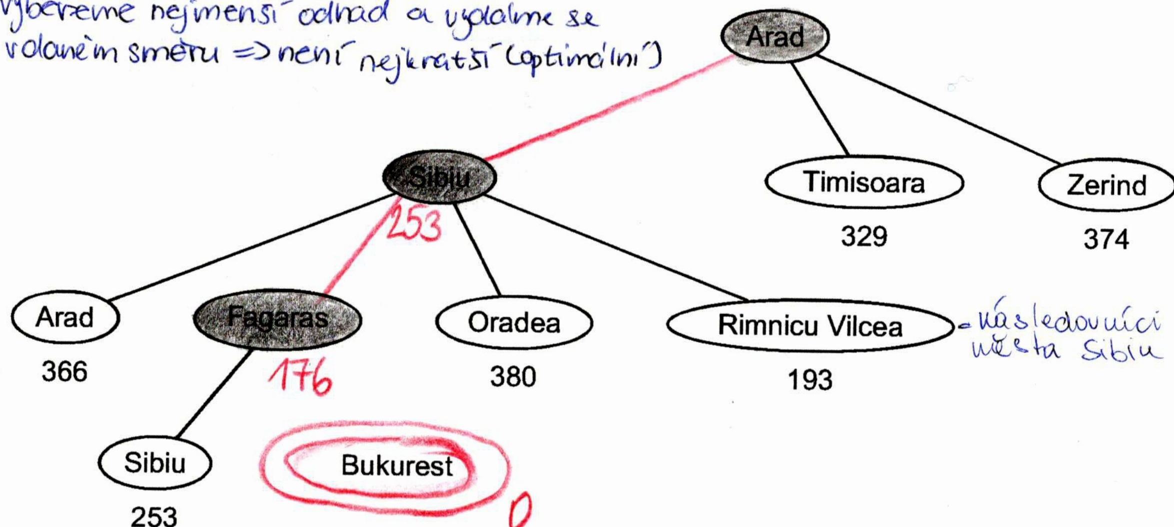
- čím menší je $h(n)$, tím blíž jsme k cíli; pokud je n cílový uzel $h(Goal) = 0$
- nejjednodušší varianta - hladové heuristické hledání (Greedy best-first search), kde $f(n) = h(n)$



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

Nalez nejkratší cestu z města Arad do Bukurešti

- hledání cesty z města Arad do Bukurešti
  - ohodnocovací fce $f(n) = h(n) = h_{Arad-Buk}(n)$, přímá vzdálenost z n do Bukurešti (vzdušná vzdálenost)



Výpočet přínosu uzlu je roven jeho odhadu vzdálenosti do cíle.

- vlastnosti:
  - expanduje vždy uzel, který se zdá nejblíže k cíli
  - cesta nalezená v příkladu (g(Arad → Sibiu → Fagaras → Bukurest) = 450) je sice úspěšná ale není optimální (g(Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bukurest) = 418)
  - obecně není úplný (může se "vydat" na nekonečnou cestu a nikdy nezkusit jiné možnosti, dostat se do cyklu (např. cesta Iasi → Fagaras - bude se neustále vracet z Iasi do Neamt a zpět)
  - není optimální - nemusí najít nejkratší cestu
  - časová složitost - $O(b^m)$, kde m je max. hloubka, s dobrou heuristikou ji může být redukována, musí si pamatovat stále celý strom
  - prostorová složitost - $O(b^m)$, drží každý uzel v paměti

Hledání nejlepší cesty - algoritmus A*

- není náročnější než hladový
- uzly ohodnocuje pomocí g(n) - cena dosažení uzlu a h(n) - cena cesty z uzlu do cíle => pro každý uzel připočítává cenu cesty, kterou jsme už ušli

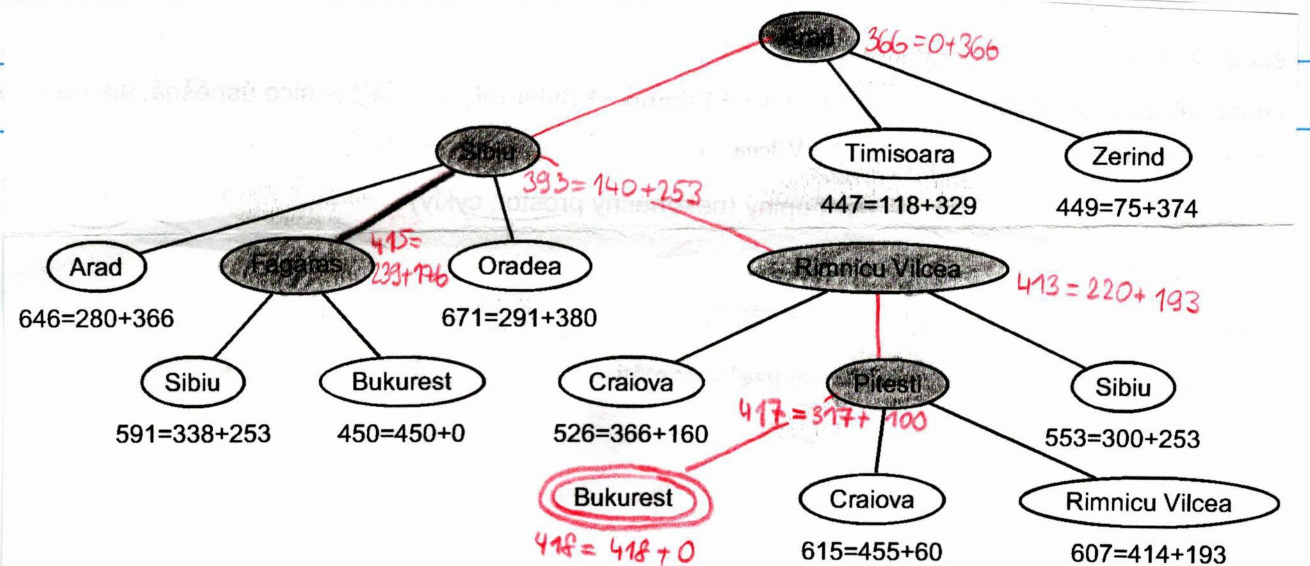
$$f(n) = g(n) + h(n)$$

  - $g(n)$ - cena cesty do n (od kořene do uzlu)
  - $h(n)$ - odhad ceny cesty z n do cíle
  - $f(n)$ - odhad ceny nejlevnější cesty, která vede přes n
- A* je optimální pokud h(n) je tzv. přípustná (admissible) heuristika, kdy h(n) není nikdy větší než skutečná cena cesty do cíle

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

=> odhad se volí vždy kratší nebo roven ceně libovolné možné cesty do cíle, např. přímá vzdálenost h_{vzd_Buk} není nikdy delší než (jakákoliv) cesta

- příklad: hledání cesty z Aradu do Bukurešti, ohodnocovací fce:
  $f(n) = g(n) + h(n) = g(n) + h_{vzd_Buk}(n)$, přímá vzdálenost z n do Bukurešti



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 např.* **biggest(9999).**

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

expand(P,l(N,_),_,_,yes,[N|P]) :- goal(N). % cíl   ; vytvoř cestu řešení
% list - generuj následníky a expanduj je v rámci Bound        uzel N má následníky
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),   - vytvoř podstromy Ts
hodnota nejlepšího -> bestf(Ts,F1), expand(P,t(N,F1/G,Ts),Bound,Tree1,Solved,Sol);Solved=never).  - N nemá žádného následníka
následníka  % 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),   % Bound1 = min(Bound,BF)
    continue(P,t(N,F/G,[T1|Ts]),Bound,Tree1,Solved1,Solved,Sol).
expand(_,t(_,_,[]), _,_,never,_) :- !. % nejsou další následovníci
expand(_,Tree,Bound,Tree,no,_) :- f(Tree,F), F>Bound. % limit
% pokrač. ->
```

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*

- expand (Path, Tree, Bound, Tree1, Solved, Solution):
  - Path – cesta mezi počátečním uzlem a podstromem Tree
  - Tree1 – Tree expandovaný v rámci Bound
  - jestliže je nalezen cíl pak Solution je řešení a Solved = yes.

```
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-hodnota uzlu
f(t(_,F/_,_),F).         } f-hodnota stromu

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*

vlastnosti:

- expanduje uzly podle $f(n) = g(n) + h(n)$: A* expanduje všechny uzly o $f(n) < C^*$
  A* expanduje ~~všechny~~ některé uzly o $f(n) = C^*$
  A* neexpanduje žádné uzly o $f(n) > C^*$

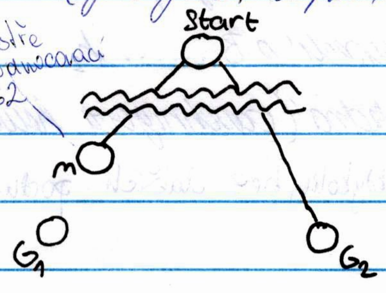
C^* – cena optimálního řešení

- je úplný (pokud [počet uzlů s $f < C^*$] $\neq \infty$)
- je optimální
- časová složitost $O((b^*)^d)$ – exponenciální v délce řešení d

  b^* – efektivní faktor větvení, viz dále
- časová složitost $O((b^*)^d)$ – drží každý uzel v paměti
- A^* může expandovat exponenciálně mnoho uzlů, zpravidla dříve než nastanou problémy s časem, vyčerpá paměť
- problém s prostorovou složitostí řeší některé nadstavbové algoritmy (např. memory-bounded heuristic search) lze nahradit o delší časovou složitost

Důkaz optimálnosti A^*

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



Pak $f(G_2) = g(G_2)$ protože $h(G_2) = 0$

$> g(G_1)$ protože G_2 je suboptimální

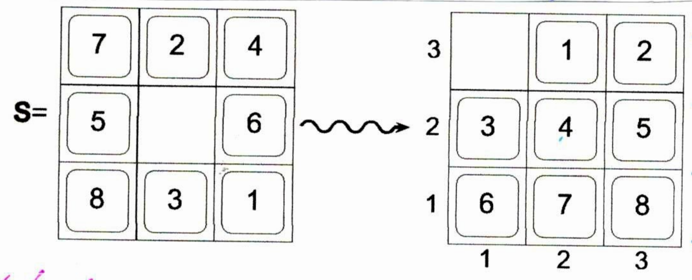
$\geq f(n)$ protože h je přípustná

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

Řešení posunovačky

- konfigurace = seznam dvojic X/Y (sloupec/řádek) = [pozice$_{díry}$, pozice$_{kamene\ 1}$, ...]

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



Volba přípustné heuristické funkce:

- $h_1(n)$ = počet dlaždiček, které nejsou na svém místě $h_1(S) = 8$
- $h_2(n)$ = součet manhattanských vzdáleností dlaždic od svých správných

pak:

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

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

Jak najít dobrou heuristiku

Jak najít přípustnou heuristickou funkci

- h_1 i h_2 jsou délky cest pro zjednodušené verze problému Posunovačka:
  - při přivolení dlaždice kamkoliv – h_1 = počet kroků nejkratšího řešení
  - při posouvání dlaždice kamkoliv o 1 pole (i na plné) – h_2 = počet kroků nejkratšího ř.

=> relaxovaný problém – problém s méně omezeními na akce než pův. problém

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

Optimální řešení pův. problému = řešení relaxovaného problému.

- posunovačka a relaxovaná posunovačka
  - dlaždice se může přesunout z A na B <=> A sousedí s B ∧ B je prázdné
  - 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 prázdné (Gaschnigova heuristika)

    c) dlaždice se může přesunout z A na B ...... h_1 (kdykoliv, bez dalších podmínek)

Určení kvality heuristiky (133)

- efektivní faktor větvení b^* – pokud N je počet všech vygenerovaných uzlů pomocí A^* a hloubka řešení je d pak b^* je efektivní faktor větvení (reálné číslo) ~~který má~~

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

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

- heuristika je tím lepší, čím je b^* blíž hodnotě 1

d	Průměrný počet uzlů IDS	$A^*(h_1)$	$A^*(h_2)$	Efektivní faktor větvení b^* 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

- měření b^* na malé množině testovacích sad – dobrá představa o přínosu heuristiky
- h2 je ve všech testováních lepší a ≥ h1
- h2 je lepší u posunovačky až zhruba do 100 × 100

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

- 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)

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

skutečný čas výpočtu

$$Fin = \max(F_j)$$

heuristická fce h

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

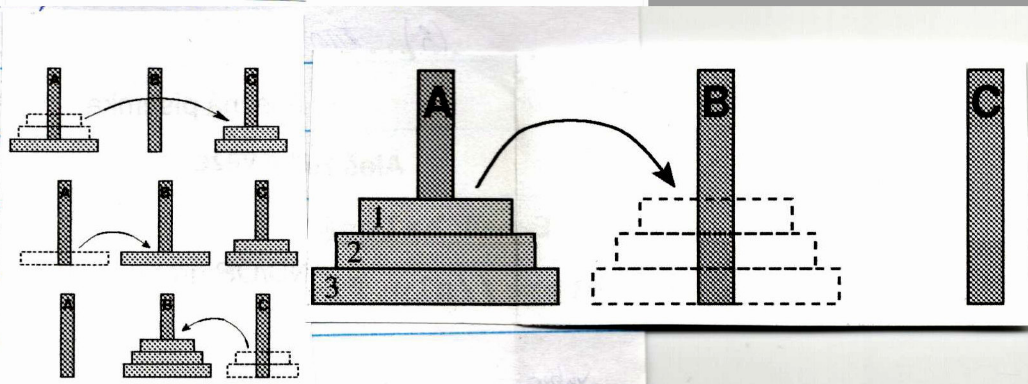
```
h(Tasks * Processors * Fin, H)  :-
   totaltime(Tasks, Tottime),
   sumnum(Processors, Ftime, N),
   Finall  is  (Tottime + Ftime)/N,
   ( Finall  > Fin, !,  H is  Finall  - Fin
     ;
     H = 0).

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).
...
```

(poznámky u kódu: totaltime – celkové trvání čekajících úloh; Ftime – součet konečných časů procesorů, N – jejich počet)

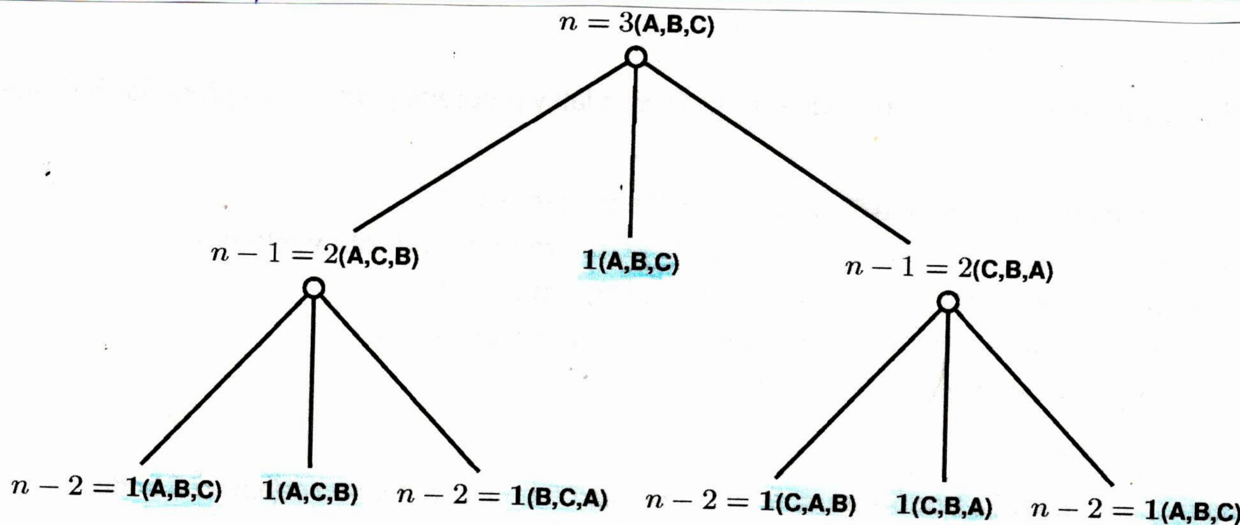
Dekompozice problému, AND/OR grafy

Hanoiské věže

- máme tři tyče A, B, 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



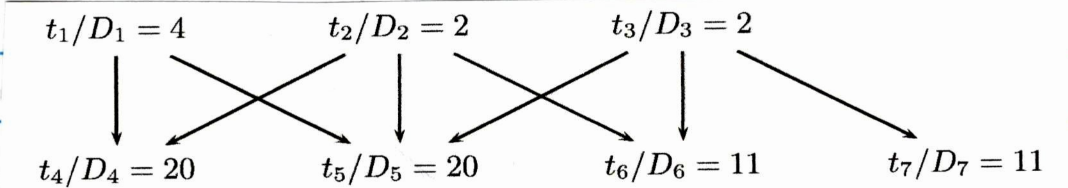
- schéma řešení pro n=3



Příklad – rozvrh práce procesorů

– úlohy t_i s potřebným časem na zpracování D_i (např.: $i = 1, \dots, 7$), m procesorů (např. $m = 3$), na nichž jsou úlohy vykonávány, každý procesor může v danou dobu vykonávat pouze 1 úkol

– relace precedence mezi úlohami – říká, které úlohy musí být kompletně splněny než jiná úloha může začít



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

	0–2	2–4	4–13	13–24	24–33
CPU_1	t_3	t_6	t_6	t_5	t_5
CPU_2	t_2	t_7	t_7	....	....
CPU_3	t_1	t_1	t_4	t_4	....

	0–2	2–4	4–13	13–24	24–33
CPU_1	t_3	t_6	t_6	t_7	....
CPU_2	t_2	..	t_5	t_5	....
CPU_3	t_1	t_1	t_4	t_4	....

– reprezentace problémové situace – potřebujeme info:

(1) tzv. seznam čekajících úloh a jejich časů zpracování

(2) současná úloha probíhající v procesoru

(3) čas ukončení schématu (částečného)

⇒ stavy: nezařazené_úlohy * zařazené_úlohy * čas_ukončení (1. seznam, 2. seznam, 3. seznam)

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

udržujeme $F1 \le F2 \le F3 \dots$

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

> **move(+Uzel, -NaslUzel, -Cena)**
> **Uzel** – *aktuální stav*
> **NaslUzel** – *nový stav*
> **Cena** – *cena přechodu*

```
move(Tasks1*[_/F|Active1]*Fin1, Tasks2*Active2*Fin2, Cost) :-
    del1(Task/D,Tasks1,Tasks2),          % vyber čekající úlohu
    not (member(T/_,Tasks2),before(T,Task)),   % ověř precedenci
    not (member(T1/F1,Active1),F<F1,before(T1,Task)),   % ověř aktivní úlohy
    Time is F+D,                          % konečný čas aktivované úlohy
    insert(Task/Time,Active1,Active2,Fin1,Fin2), Cost is Fin2-Fin1.
move(Tasks*[_/F|Active1]*Fin,Tasks*Active2*Fin,0) :- insertidle(F,Active1,Active2).   % nech procesor nečinný

% úloha T1 před T2 podle precedence
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,!.   % uspořádání seznamu úloh
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,!.   % ponech procesor nečinný až do prvního většího konečného času
insertidle(A,[T/B|L],[T/B|L1]) :- insertidle(A,L,L1).

goal([]*_*_).   % žádný čekající úkol
```

> **before(+Task1, +Task2)**
> *tranzitivní obal relace* **precedence**

```
?- 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
```

- hanoi(N,A,B,C,Moves) - je pravdivé pokud Moves je sekvence pohybů vyžadovaných k přesunu N disků z tyče A na tyč B s pomocí C podle daných pravidel

And/Or grafy

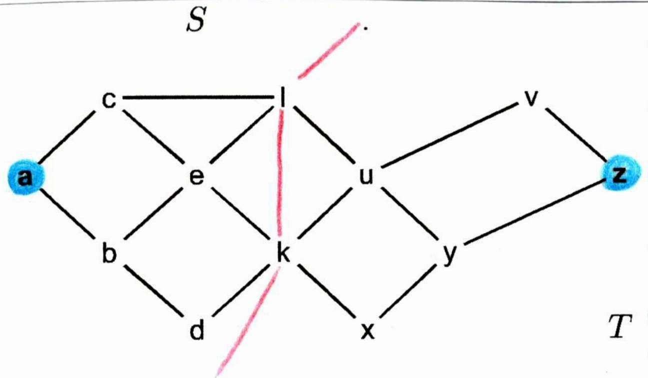
města: **a**, ..., **e** ... ve státě S

l a **k** ... hraniční přechody

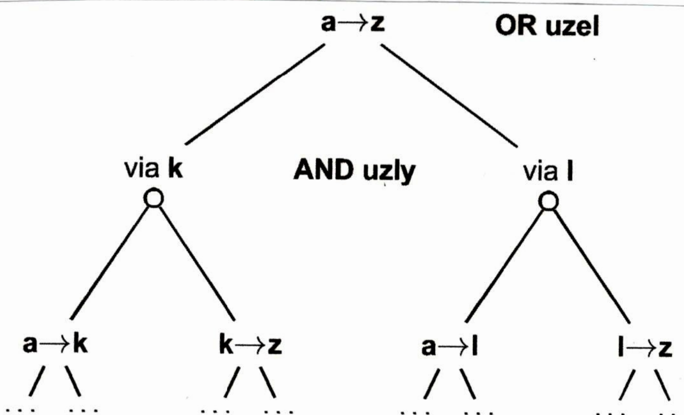
u, ..., **z** ... ve státě T

hledáme cestu z **a** do **z**:

- ➜ cesta z **a** do hraničního přechodu
- ➜ cesta z hraničního přechodu do **z**



- můžeme jít buď skrz l nebo skrz k -> původní problém můžeme rozdělit na dva podproblémy - každý z nich může být řešen nezávisle na druhém



- 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.

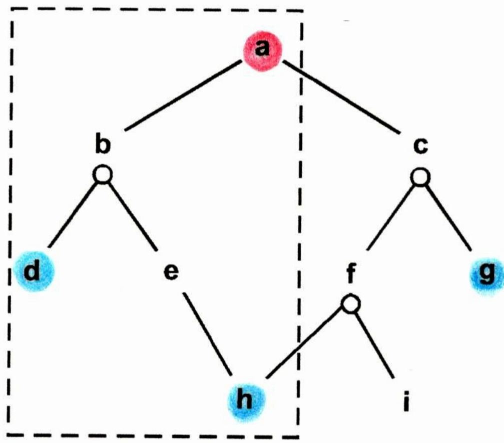
AND uzel x s následníky y1, y2, ... yM: x :- y1, y2, ..., yM.

cílový uzel g (elementární podproblém): g.

kořenový uzel root : ?- root.

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

Triviální prohledávání AND/OR grafu v Prologu

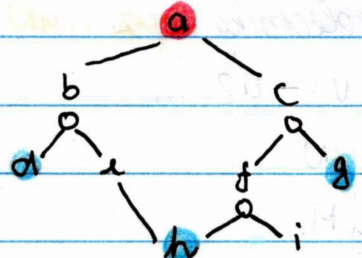



```
a :- b.        a je OR uzel se dvěma následovníky, b a c
a :- c.
b :- d, e.     b je AND uzel se dvěma následovníky d a e
e :- h.
c :- f, g.
f :- h, i.
d.
g.             d, g, h jsou cílové uzly
h.

?- a.          ptáme se, zda může být problém vyřešen
  Yes
```

- program je jednoduchý ale negeneruje strom řešení (pouze ano/ne jestli existuje), nemůžeme v něm uplatnit ceny; v případě, že by graf obsahoval cyklus, Prolog by se mohl dostat s DFS do nekonečné rekurzivní smyčky

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 potomků
  - OR uzel se chová jako běžný uzel klasického grafu
- operátory ---> a :   ?-op(600, xfx, --->)   op(+Priorita, +Typ, +Jméno)
  ?-op(500, xfx, :)
  - Priorita – číslo mezi 0 .. 1200
  - Typ – jedno z xf, yf, xfx, xfy, yfx, yfy, fy, fx
  - Jméno – funktor nebo symbol
- zápis AND/OR grafu



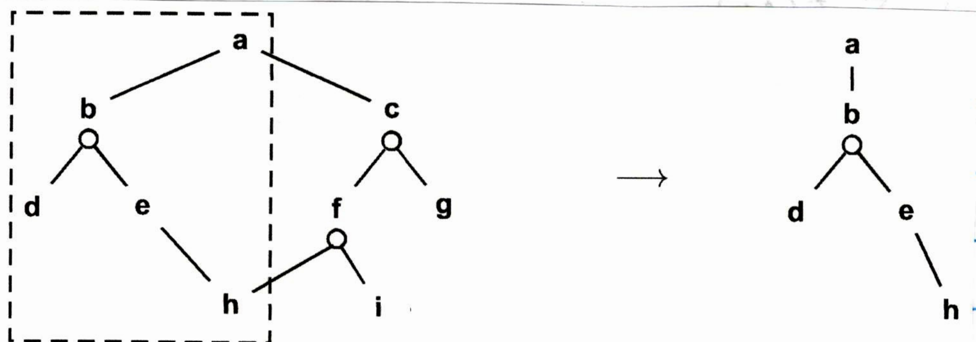

```
a ---> or:[b,c]            goal(d)
b ---> and:[d,e]           goal(g)
c ---> and:[f,g]           goal(h)
e ---> or:[h]
f ---> and:[h,i]
```

(zpracování – zápis)

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 ⇒ právě jeden z jeho následníků se svým stromem

řešení je v T
- jestliže P je AND uzel grafu G => všichni jeho následníci a jejich stromy řešení jsou v T
- každý list stromu řešení T je cílovým uzlem v G




```
% 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
```

uzel je OR uzel -> vyber následníka uzlu a jeho strom (podstrom grafu)

uzel je AND uzel -> vyřeš všechny následníky uzlu a jejich stromy (podstromy)

- pokud je uzel cílovým uzlem, pak řešením je uzel sám
- pokud je uzel OR uzel, řešení má formu Node –––> Subtree, kde Subtree je strom řešení následníka uzlu
- pokud je uzel AND uzel, pak řešení má formu Node –––> and: Subtrees, kde Subtrees je seznam stromů řešení následníků uzlu

Heuristické prohledávání AND/OR grafu

- reprezentaci doplníme o cenu přechodové hrany (=odhad obtížnosti podproblému)

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

- cenu uzlu definujeme jako cenu optimálního řešení jeho problému, chceme najít minimálně drahý strom řešení
- pro každý uzel N máme tedy odhad jeho ceny:

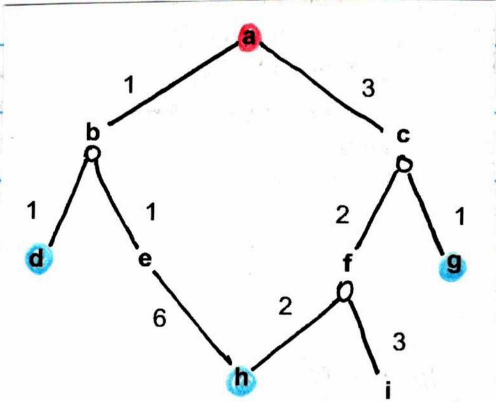
  h(N) = heuristický odhad ceny optimálního podgrafu s kořenem N

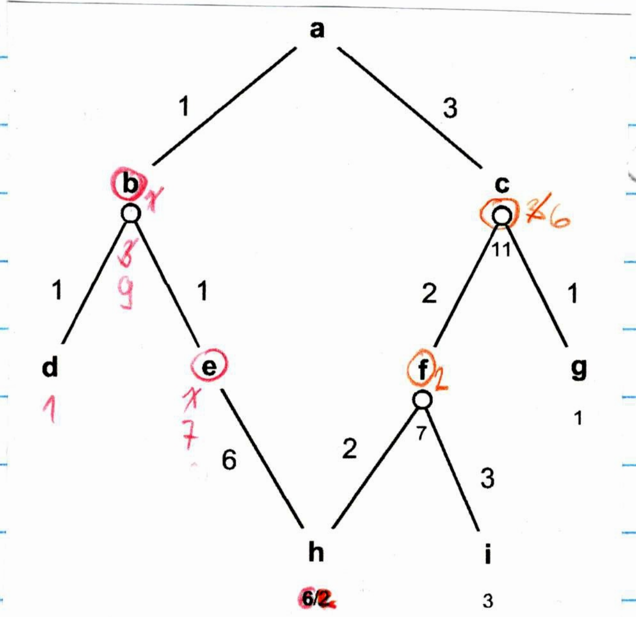
- pro každý uzel N, jeho následníky $N_1 \dots, N_b$ a jeho předchůdce M definujeme

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

Pro optimální strom řešení S je tedy $F(S)$ právě cena tohoto řešení (= suma všech hran z S)

- příklad:
   - setříděný seznam částečně expandovaných grafů
     [Nevyřešený$_1$, Nevyřešený$_2$ ..., Vyřešený$_1$ ...]
     $F_{Nevyřešený_1} \leq F_{Nevyřešený_2} \leq \dots$





Reprezentace AND/OR grafu při heuristickém prohledávání

- list and/or grafu ... struktura leaf(N, F, C) ⟶ C ... cena hrany do uzlu N
  $F = C + h(N)$ F ... příslušná heur. F-hodnota uzlu N
- OR uzel AND/OR grafu ... struktura tree(N, F, C, or: [T1, T2, T3, ...]) N ... identifikátor uzlu
  $F = C + min_i F_i$
- AND uzel AND/OR grafu ... struktura tree(N, F, C, and: [T1, T2, T3, ...])
  $F = C + \sum_i F_i$
- vyřešený list AND/OR grafu ... struktura solvedleaf(N, F)
  $F = C$
- vyřešený uzel OR AND/OR grafu ... struktura solvedtree(N, F, T)
  $F = C + F_1$
- vyřešený AND uzel AND/OR grafu ... struktura solvedtree(N, F, and: [T1, T2, ...])
  $F = C + \sum_i F_i$

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

% 1: limit Bound překročen (ve všech dalších klauzulích platí F =< Bound)
expand(Tree,Bound,Tree,no) :- f(Tree,F),F>Bound,!.
% 2: nalezen cíl
expand(leaf(Node,F,C),_,solvedleaf(Node,F),yes) :- goal(Node),!.
% 3: expanze listu
expand(leaf(Node,F,C),Bound,NewTree,Solved) :- expandnode(Node,C,Tree1),!,
    (expand(Tree1,Bound,NewTree,Solved);Solved=never,!).
% 4: expanze stromu
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) :-
    bestf(SubTrees,H), F is C+H,!.
continue(never,_,_,_,_,_,never) :- !.
continue(no,Node,C,SubTrees,Bound,NewTree,Solved) :- bestf(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 expanduje 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

```
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,
    expandsucc(Successors,SubTrees),bestf(Op:SubTrees,H),F is C+H.
expandsucc([],[]).
expandsucc([Node/C|NodesCosts],Trees) :- h(Node,H),F is C+H,expandsucc(NodesCosts,Trees1),
    insert(leaf(Node,F,C),Trees1,Trees).

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

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

combine(OtherTrees,NewTree, Solved1,NewTrees,Solved) kombinuje výsledky expanze stromu a seznamu stromů

expandnode převede uzel z **Node—>AndOr:S** do **tree(Node,F,C,SS)**

allsolved zkontroluje, jestli všechny stromy v seznamu jsou vyřešené

```
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).

% první následovník v OR-uzlu je nejlepší
bestf(or:[Tree|_],F) :- f(Tree,F),!.
bestf(and:[],0) :- !.
bestf(and:[Tree1|Trees],F) :- f(Tree1,F1),bestf(and:Trees,F2),F is F1+F2,!.
bestf(Tree,F) :- f(Tree,F).

selecttree(Op:[Tree],Tree,Op:[],Bound,Bound) :- !.  % The only candidate
selecttree(Op:[Tree|Trees],Tree,Op:Trees,Bound,Bound1) :- bestf(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).
```

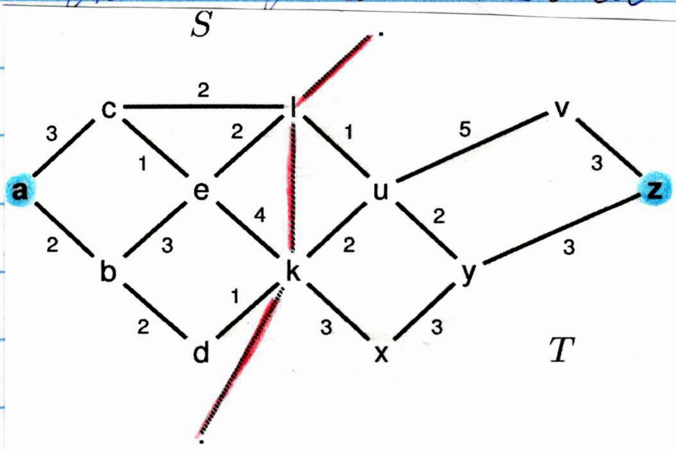
insert vkládá strom do seznamu stromů se zachováním třídění

bestf 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) - chceme-li najít cestu z Mesto1 do Mesto2, měli bychom nalézt cesty přes 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)).
```

- vlastní hledání cesty
  1) existují-li klíčové body Y1 a Y2 mezi městy A a Z, pak hledej jednu z cest:
     - z A do Z přes Y1
     - z A do Z přes Y2
  2) není-li mezi městy klíčové město ⇒ hledej souseda Y města A takového, že existuje cesta z Y do Z

- konstrukce příslušného AND/OR grafu

```
?- op(560,xfx,via).   % operátory X-Z a X-Z via Y
```
má nižší prioritu než '-' a než '--->'

```
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). ...
```

pravidlo expanze pro problém X-Z když nejsou mezi X a Z žádné klíčové body, ceny všech spojnic = 0

```
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 */
```
redukuj via problém na dva AND problémy
jít z X do X je triviální

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

• úplný (pro konečné b), optimální (podle délky cesty), není optimální podle ceny cesty, č.s. $O(b^{d+1})$, $O(b^{d+1})$ p.s. (drží všechny navštívené uzly v paměti)

Prohledávání podle ceny - úplný (pro cenu akce ≥ ε), optimální (pro cenu ≥ ε), č.s. $O(b^{1+[C^*/\varepsilon]})$, p.s. $O(b^{1+[C^*/\varepsilon]})$

Prohledávání s postupným prohlubováním - IDS - úplný pro konečné b, optimální pro cenu g(n) závislou na hloubce, č.s. $O(b^d)$, p.s. $O(bd)$

Informované prohledávání stavového prostoru - má info o odhadu blízkosti stavu k cílovému s. ⇒ heuristická fce h(n)

Heuristické hledání nejlepší cesty

• ohodnocovací fce f(n) - pro každý uzel (měří přínos uzlu)

• heuristická fce h(n) - odhad vzdálenosti uzlu od cíle, h(Goal) = 0

Hladové heuristické hledání - f(n) = h(n), vybírá uzel s nejmenším odhadem vzdálenosti (zdá se nejblíže cíli) do cíle ⇒ není optimální (nemusí najít nejkratší cestu); není úplný (nekonečné cesty, cykly), č.s. $O(b^m)$, p.s. $O(b^m)$ - udržuje v paměti každý uzel

Algoritmus A*

• f(n) = h(n) + g(n); g(n) - cena cesty do n

• je optimální pokud $0 \le h(n) \le h^*(n)$ - skutečná cena cesty do cíle ⇒ přípustná heuristika

• prologovský zápis - pro list generuje následníky a expanduje v rámci Bound

- pro tree, fc Bound generuje nejslibnější podstrom, pokrač. dle výsledku

- succlist - setřídí seznam ~~hodnot~~ listů podle f-hodnot

- continue - volí zp. pokračování podle výsledku expand

- bestf - volí nejlepší f hodnotu ze seznamu stromů

• expanduje všechny uzly s $f(n) < C^*$, některé uzly s $f(n) = C^*$, a žádné u. s $f(n) > C^*$

• úplný (pokud [počet uzlů s $f(n) < C^*$] ≠ ∞), optimální, č.s. $O((b^*)^d)$, p.s. $O((b^*)^d)$

Relaxovaný problém - problém s menší omezeními než má pův. problém

• cena optimálního řešení relaxovaného problému je přípustná heuristika pro pův. problém

Efektivní faktor větvení b^* - čím blíž je jeho hodnota 1, tím je heuristika lepší

Rozvrh práce procesorů

• relace precedence - které úlohy musí být splněny, než jiná může začít

• stavy: nezařazené úlohy * zařazené úlohy * čas ukončení

• výběr čekající úlohu → ověř precedenci → ověř aktivní úlohy ⇒ nech procesor nečinný

Hanojské věže

```
?- 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).
```

AND/OR grafy - rozklad problému na podproblémy

• AND uzel vyžaduje průchod všech svých potomků, s nimi následovníci jsou součástí řešení (se svými podstromy)

• OR uzel se chová jako běžný uzel klasického grafu, pouze 1 z nich a jeho podstrom jsou součástí řešení

• listy jsou cílové uzly

```
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)
```

Heuristické prohledávání AND/OR grafu

• doplnění ceny přechodové hrany, $h(N)$ = odhad ceny optimálního podgrafu s kořenem N

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

• expand - expanduje Tree po Bound, výsledkem je NewTree se stavem Solved

• expandlist - expanduje 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

• combine - kombinuje výsledky expanze stromu a seznamu stromů

• expandnode - převede uzel z Node → AndOr:S do tree(Node, F, C, SS) → vytvoří strom z uzlu a jeho následníků

• allsolved - zkontroluje jestli všechny stromy v seznamu jsou vyřešené

• insert - vkládá strom do seznamu stromů se zachováním třídění

Problem osmi dam:

```
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).
```

Prohledávání stavového prostoru: statické a deterministické prostředí, mění se jen po zásahu z vnějšku, ze znalosti prostředí a vykonávané akce lze predikovat následný stav

- b - faktor větvení, max. počet následníků
- d - hloubka cíle
- m - maximální hloubka větve/cesty

Neinformované prohledávání: zná pouze popis úlohy (počáteční/cílový stav, přechodová fce), rozpozná cílový stav, žádná dodatečná info

Prohledávání do hloubky:

```
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).
```

- není úplné (nekonečná větev), není optimální, č.s. $O(b^m)$, p.s. $O(b^m)$ - uprodované uzly může odstranit z paměti

Prohledávání do hloubky s limitem

```
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)
```

- není úplné (pro $l<d$), není optimální (když $l>d$), č.s. $O(b^l)$, p.s. $O(bl)$

Prohledávání do šířky

```
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),!,
    bread_first_search(Path1,Solution); bread_first_search(Paths,Solution).
```

• vícesměrný predikát pro přidávání/odebírání

```
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)).
```

Reprezentace grafů - soubor vrcholů a hran

• orientovaný graf: `graph(V,E), e(V1,V2)`

• orientovaný ohodnocený graf: `vgraph(V,E), e(PocatecniV, KoncovyV, Cena)`

• neorientovaný graf: fakty + pravidla, nelze přímo zapsat

• cesta v grafu - nesmí obsahovat žádný cyklus; neorientovaný graf:

```
path(A,Z,Graf,Cesta):- path1(A,[Z],Graf,Cesta).
path1(A,[A|Cesta1],_,[A|Cesta1]).
path1(A,[Y|Cesta1],Graf,Cesta):- adjacent(X,Y,Graf), not(member(X,Cesta1)),
        path1(A,[X,Y|Cesta1],Graf,Cesta).
```

neorientovaný ohodnocený graf:

```
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).
```

• kostra grafu - souvislý strom procházející všechny vrcholy grafu, který neobsahuje žádný cyklus

```
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)).
```

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

• rozdílové seznamy – Seznam2 jako volná proměnná slouží jako ukazatel na konec Seznamu

```
append_dl(A-B, B-C, A-C).
```

• třídění seznamů

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

qsort_dl(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]).
```

Binární stromy: orientovaný graf s jedním vrcholem (kořenem), z něhož existuje cesta do všech vrcholů grafu t(L, Hodn, P)

• přidávání do binárního stromu

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

• odebírání z binárního stromu

```
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)
```

Principy prologu: backtracking řízený unifikací, hojně využívá rekurzi

- Cíl je dokázán, unifikuje-li se s hlavou nějaké klauzule a všechny podcíle v těle klauzule jsou také dokázány.
- Fakty jsou vždy pravdivé, pravidla jsou pravdivá, pokud je splněna nějaká podmínka.

Seznam: rekurzivní datová struktura - každá položka odkazuje na položku stejného typu

- prvek seznamu

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

- smazání prvku seznamu

```
del(_,[],[]).
del(A,[A|T],V) :- del(A,T,V).
del(A,[H|T1],[H|T2]) :- A\=H, del(A,T1,T2).

del1(A,[A|T],T).
del1(A,[H|T1],[H|T2]) :- del1(A,T1,T2).
```

- rozšíření seznamu o daný prvek

```
insert(A,L,[A|L]).                        insert1(X,List,[X|List]).
insert(A,[H|T1],[H|T2]) :- insert(A,T1,T2).
```

- permutace

```
perm1([],[]).
perm1([H|T],L) :- perm1(T,V), insert(H,V,L).

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

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

- spojení seznamů - vícesměrný predikát

- bestf - vyhledá uloženou hodnotu AND/OR stromu/uzlu
- selecttree - vybere BestTree z Trees, zbytek je v OtherTrees, Bound je závora pro Trees, Bound1 pro BestTree
- najdeme vždy optimální řešení pokud $h(n) \leq h^*(n)$

Problémy s omezujícími podmínkami - CSP

- řešením je úplné (každá proměnná X) konzistentní (neporušuje žádné omezení) přiřazení hodnot proměnným
- diskrétní hodnoty proměnných - konečné domény a nekonečné d. (nelze vyjmenovat všechna možná přiřazení; číselné lineární problémy jsou řešitelné, nelineární obecně ř. nemají)
- spojité hodnoty proměnných (reálné problémy), lin. omezení jsou řešitelné v polynom. čase
- preferenční omezení - hledání optimalizovaného řešení vzhledem k cenč přiřazení

Prohledávání s navracením - backtracking, DFS pro CSP, neinformovaná strategie, v každém kroku přiřazení jedné proměnné => d^n listů, přiřazení jen konzistentní

Inkrementální formulace CSP umožňuje CSP převést na standardní prohledávání, prohl. strom odpovídá hloubce n (počet proměnných), řešení v této hloubce (úplné přiřazení)

Ovlivnění efektivity prohl. s navracením

- nejomezenější proměnná, nejvíce omezující proměnná, nejméně omezující hodnota
- dopředná kontrola (seznam možných hodnot pro zbývající proměnné)
- propagace omezení (kontrola nekonzistence ve zbývajících proměnných)

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

Multiagentní prostředí - soupeřící agenti - prvek náhody; hry - deterministické MP - 2 agenti

- algoritmy soupeřícího prohledávání -> strategie, lokálně optimální řešení (čas. limit)

Algoritmus MINIMAX

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

- úplný pouze pro konečné stromy, optimální proti optimálnímu oponentovi, č.s. $O(b^m)$ - musím projít celý strom, p.s. $O(bm)$ - prohl. do hloubky

• časové omezení lze řešit ohodnocovací funkcí (odhadem přínosu pozice) nebo pomocí ořezávacího testu

α-β prořezávání – efektivnější varianta minimaxu, odřezává uzly, které nemohou ovlivnit konečné rozhodnutí; výsledek je stejný jako u minimaxu

– dobré uspořádání možných tahů ovlivní efektivitu prořezávání

– č.s. $O(b^{m/2})$ – zdvojí hloubku prohledávání

– α – nejlepší hodnota pro MAXe, odřezáváme stavy $V(P) \leq \alpha$

– β – dosud nejlepší hodnota pro MINa, odřezáváme stavy $V(P) \geq \beta$

– boundtedbest – postupně propočítává jednotlivé pozice a hledá takovou, aby byla dobrá vzhledem k α, β

– newbounds – vypočítává nové hranice (mohou se jen zužovat)

minimax_cutoff – stejný jako minimax, kromě ořezávacího testu a ohodnocovací fce

ohodnocovací fce – Eval(s) – odhad perspektivnosti hry ze zadané pozice, řadí konečné stavy podle přínosnosti. Chová se stejně pro libovolnou monotónní transformaci fce Eval

• konečné stavy řadí podle přínosnosti – záleží pouze na uspořádání – ohodnocení jako ordinální fce

nedeterministické hry – prvek náhody

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

– ořezávání v nedeterministických hrách je větší, pokud je možné dopředu stanovit limity ohodnocení listů

– hody kostkou zvyšují b, s vyšší hloubkou klesá pravděpodobnost dosažení zvoleného uzlu
  => α-β prořezávání je mnohem méně efektivní

– pořadí zachovávající transformace mění nejlepší tah, Eval u nedeterministických her by měla proporciálně odpovídat očekávanému výnosu

  • chování je zachováno pouze pro pozitivní lin. transformaci Eval

hry s nepřesnými znalostmi – výpočet pravděpodobnosti každého možného rozdání, prohledávání domnělého stavového prostoru

Logický agent - využívá znalosti
- reprezentace znalostí
- vyvozování znalostí = inference
- deklarativní i procedurální přístup - co má udělat i jak to má udělat
- komponenty - inferenční stroj a báze znalostí (KB)
- akce logického agenta:

```
kb_agent_action (KB, ATime, Percept, Action, NewATime):-
    make_percept_sentence (Percept, ATime, Sentence),
    tell(KB, Sentence),
    make_action_query (ATime, Query),
    ask (KB, Query, Action),
    make_action_sentence (Action, ATime, ASentence),
    tell (KB, ASentence),
    NewATime is ATime + 1.
```

Popis světa - PEAS - míra výkonnosti, prostředí, akční prvky, senzory

Vlastnosti Wumpusovy jeskyně - jen lokální vnímání, deterministické, sekvenční na úrovni akcí, statické, diskrétní, Wumpus jako vlastnost prostředí ne jako agent

Důsledek - jedna věc logicky vyplývá z druhé - $KB \models \alpha$

Model - formální strukturovaný (abstraktní) svět, umožňuje vyhodnocení pravdivosti
- kontrola modelů - kontrola, zda je α je true ve všech modelech, v nichž KB je true

Inference - vyvozování požadovaných důsledků - $KB \vdash_i \alpha$

i je bezesporná $\iff \forall KB \vdash_i \alpha \implies KB \models \alpha$

i je úplná $\iff \forall KB \models \alpha \implies KB \vdash_i \alpha$

- jestliže máme sémantiku "pravdivou" o reálném světě, můžeme vyvozovat závěry o skutečném světě pomocí logiky

Výroková logika
- každý model musí určit pravdivostní hodnoty výrokových symbolů
- dva výroky jsou logicky ekvivalentní právě tehdy, když jsou pravdivé ve stejných modelech

$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha))$ $\neg(\alpha \land \beta) \equiv (\neg\alpha \land \neg\beta)$

$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \lor \beta)$ $\neg(\alpha \lor \beta) \equiv (\neg\alpha \land \neg\beta)$

$(\alpha \land (\beta \lor \gamma)) \equiv ((\alpha \land \beta) \lor (\alpha \land \gamma))$ $(\alpha \lor (\beta \land \gamma)) \equiv ((\alpha \lor \beta) \land (\alpha \lor \gamma))$

- výrok je platný $\Leftrightarrow$ pravdivý ve všech modelech
- výrok je splnitelný $\Leftrightarrow$ je pravdivý v některých modelech
- výrok je nesplnitelný $\Leftrightarrow$ je nepravdivý ve všech modelech
- věta o dedukci: $KB \models \alpha \Leftrightarrow (KB \Rightarrow \alpha)$ je platný výrok
- inference pomocí důk. sporem: $KB \models \alpha \Leftrightarrow (KB \land \neg\alpha)$ je nesplnitelný

Důkazové metody - kontrola modelů, aplikace inferenčních pravidel

Inference kontrolou modelů - kontrola všech modelů do hloubky je bezesporná a úplná $O(2^n)$, NP-úplný problém (pro n symbolů)

Hornova klauzule: disjunkce literálů (nejvýše 1 pozitivní), lze také jako konjunkce

= { výrokový symbol / (konjunkce symbolů) $\Rightarrow$ symbol

H. formule - konj. norm. forma z H. klauzulí.

-inference - algoritmy dopředného a zpětného řetězení (č. s. lineární)

Dopředné řetězení - je řízeno daty, může udělat hodně nadbytečné práce bez ustálení cíle

Zpětné řetězení - je řízeno dotazem, vhodné pro odpověď na konkrétní dotaz, složitost může být mnohem větší než lineární (kontroluje jestli daný podcíl už nebyl řešen - cykly), dokazuje všechny premisy pravidla s q jako důsledkem

Rezoluce - obecný inferenční algoritmus

- konj. norm. forma
- bezesporná a úplná pro výrokovou logiku

Výroková logika je deklarativní (syntaxe přímo koresponduje s fakty)

- umožňuje zpracovávat částečné, disjunktivní, negované informace
- je kompoziční (význam věty je odvozen z významů jejích částí)
- význam je kontextově nezávislý
- má velice omezenou expresivitu

Predikátová logika: objekty, relace, funkce

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

- $\exists x\, P$ je pravdivé v modelu m $\Leftrightarrow$ P je pravdivá pro x = nějaký objekt z modelu m
- inference je možná pouze podle inferenčních pravidel (dopředné/zpětné ř., rezoluce...)
- zákl. inferenční pravidlo - zobecněné Modus Ponens (A platí, A => B platí, pak i B platí)
  - hledáme substituce potřebné pro odvozování, hledáme substit., která zajistí, že dva logické výrazy budou identické
- diagnostické pravidlo (odvodí příčiny z následku) x příčinné p. (vyjadřuje z premisy)

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

Logická analýza PJ - analýza významu výrazů (vět) PJ

- pojem ≠ výraz, pojem ≠ představa

Nedostatečná expresivita PL1 vlastnost červený ≠ červená barva (nelze vyjádřit)

Extensionalismus

- význam výrazu se nemění (nezávisí na světě a čase), ale mění se jeho hodnota (v závislosti na světě a čase)
- intenze - objekty typu funkcí, jejichž hodnoty závisí na světě a čase
- extenze - objekty na světě a čase nezávislé

Transparentní intenzionální logika - logický systém pro zachycení významů výrazů PJ

- nositel významu (konstrukce) není prvkem formálního aparátu, tento aparát pouze studuje konst.
- zachycení intenzionality je přesně popsáno z mat. hlediska
- typy objektů - typová báze = $\{o, \iota, \tau, \omega\}$, funkce nad typovou bází, typy vyšších řádů
  - $\alpha_{\tau\omega}$ - intenze (závislost na čase a světě)

Možný svět v TIL - rozhodovací systém, pro ∀ prvek intenzionální báze obsahuje konzistentní přiřazení hodnot

Intenzionální sestup - identifikace extenze pomocí intenze, světa w, času t, aplikace na svět a čas

Předpoklad uzavřeného světa - cokoliv, o čem nevíme, že je pravda, bereme jako nepravdu

Předpoklad jednoznačných pojmenování - různá jména označují různé objekty

Rezoluce je důkaz sporem, dokazuje se, že pro $KB \models \alpha$ je $KB \wedge \neg\alpha$ nesplnitelné, používá $KB, \neg\alpha$ v CNF

- postup důkazu F: $\neg F$ -> rozšiřujeme s KB až do odvození prázdné klauzule - spor => F musí platit

Extralogické informace

- objekty reálného světa mají mezi sebou vztahy
- stav světa se může měnit v čase
- ne každá informace je zaznačena

Sémantické sítě – ve formě grafu reprezentují faktové znalosti (pojmy + vztahy)

- podtřída – vztah mezi třídami
- instance – vztah mezi konkrétním objektem a jeho nadřazenou třídou
- jestliže určitá vlastnost platí pro třídu, platí i pro všechny její podtřídy
- — " — , platí i pro všechny prvky této třídy
- ALE mechanismus vzorů (hodnota vlastnosti u třídy, platí ta nejbližší objektu) a výjimek (u konkrétních objektů lišících se od vzoru)

  → víra v hodnotu vlastnosti objektu se může měnit s příchodem nových informací o klasifikaci objektu

Rámce – varianta sémantických sítí; obsahuje objekty: , sloty : hodnota slotu

Pravidlové systémy – pravidla IF , THEN (podmínka, akce) zachycující znalosti experta

- znalosti mohou být strukturovány do modulů
- systém může být rozšířen o nová pravidla bez změny zbytku systému

Expertní systémy – aplikace pravidlových systémů

Metody pro práci s nejistotou

- defaultní /nemonotónní logika – X bude OK, pokud se nenajde protipříklad
- pravidla s faktory nejistoty – $A_5 \xrightarrow[0,3]{}$ dostat se včas
- pravděpodobnost – vztah mezi tvrzením a jeho pravdivostí vzhledem k podmínkám, může se měnit s novými podmínkami

Vyvozování s nejistými znalostmi

- používá náhodné proměnné – fce přiřazující hodnoty jevům nebo výsledky měření sledovaného jevu
- distribuce pravd. náh. prom. – vektor pravd., že daná náh. prom. bude mít určitou hodnotu
- výpočet logiky neslučitelných událostí: $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$

Bayesovské pravidlo:
- podmíněná pravděpodobnost - $P(a|b) = \frac{P(a \land b)}{P(b)}$ if $P(b) \neq 0$ ⟹
- určení diagnostické pravděpodobnosti ze znalosti příčinné pravděpodobnosti:

$$P(\text{Příčina}|\text{Následek}) = \frac{P(\text{Následek}|\text{Příčina})P(\text{Příčina})}{P(\text{Následek})}$$

Učení modifikuje rozhodovací systém agenta pro zlepšení jeho výkonnosti, využívá jeho vjemy z prostředí, složky:
- výkonnostní komponenta, kritika, komponenta učení, generátor problémů
- senzory, akční prvky

Komponenta učení - závisí na typu výkonnostní komponenty a její části, která má být učena, jak je tato část reprezentována a jaká zpětná vazba je k dispozici
- učení s dohledem - učení funkcí z příkladů vstupů a výstupů
- učení bez dohledu - učení vzorů na vstupu vzhledem k reálnému prostředí
- posílené učení - podle odměn/pokut

Induktivní učení - učení funkcí z příkladů, hledá hypotézu $h \approx f$, dobrá h bude dobře zobecnitelná (schopná předpovídat neznámé příklady správně)
- h je konzistentní ⟺ souhlasí s f na všech trénovacích příkladech (pravidlo Occamovy břitvy - nejjednodušší řešení je nejlepší)
- protichůdné požadavky na prostor hypotéz - pokrýt co největší množství hledaných fcí, udržet nízkou výpočetní složitost hypotézy (může být výhodnější povolit nekonzistentní h)

Rozhodovací stromy - vyjádří libovolnou Booleovskou fci vstupních atributů - odpovídá výrokové logice

Prostor hypotéz s větší expresivitou zvyšuje šance na nalezení přesného vyjádření cílové funkce, ale zvyšuje i počet možných konzistentních hypotéz ⟹ můžeme získat nižší kvalitu generalizace

Triviální konstrukce rozhodovacího stromu - každý příklad trénovací sady má 1 cestu od kořene k listu, přesně bude fungovat pouze na stejných příkladech jako v trénovací sadě - negeneralizuje vzory, jen kopíruje pozorování

Heuristická konstrukce kompaktního stromu - nejmenší r. s., který souhlasí s příklady je příliš složité najít ⟹ dostatečně malý (výběr atributů v co nejlepším pořadí)

Výběr atributu - dobrý atribut rozdělí příklad na podmnožiny, které jsou (nejlépe)

všechny pozitivní nebo všechny negativní

- míra informace – čím méně dopředu víme o výsledku obsaženém v odpovědi, tím více informace je v ní obsaženo
    - 1 bit = odpověď na Booleovskou otázku s $\langle P(T) = \frac{1}{2}, P(F) = \frac{1}{2} \rangle$

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

    - algoritmus IDT – učení formou rozhodovacích stromů
    - tvar křivky učení závisí na (ne)realizovatelnosti fce (chybějící atributy, omezený prostor hypotéz) a na expresivitě (nadbytečné množství nerelevantních atributů)

Neuronové sítě

- jednotky neuronové sítě jsou propojeny vazbami o číselné váze $W_{j,i}$; aktivace a_j jednotky j
- g – aktivační fce (musí být nelineární (aby síť nebyla lineární)), aktivuje jednotku pro pozitivní příklady (+1), jinak neaktivní (0); prahová fce / sigmoida
- neuronová j. umí implementovat zákl. Booleovské fce (AND, OR, NOT)

Síť s předním vstupem = parametrizovaná nelineární fce vstupu, necyklická, nemá vnitřní paměť

Rekurentní sítě – cyklické (výstup na vstup) podporují paměť, složitější a obtížnější

Perceptron – jednovrstvá síť, reprezentuje lineární separátor (nadrovina) na vstupu

- může reprezentovat pouze lineárně separabilní fce
- učí se upravováním vah tak, aby se snížila chyba na trénovací sadě (pro poz. příklady se zvýší, pro neg. sníží), dokud nedosáhne ukončovacího kritéria

Vícevrstvé neuronové sítě

- skryté jednotky – s jednou skrytou vrstvou vyjádří všechny spojité fce, se dvěma pak všechny fce => zvyšují prostor hypotéz, které můžeme reprezentovat
- problémy – dosažení lokálního maxima, pomalá konvergence, upnutí na příklady

=> perceptrony mají nízkou vyjadřovací sílu, vícevrstvé sítě jsou dostatečně silné a mohou být trénovány pomocí zpětného šíření chyby

Řečové akty – mluvčí -> promluva -> posluchač

Komunikační fáze záměr -> generování -> syntéza -> vnímání -> analýza -> vyjasňování -> zahrnutí do KB

Gramatiky

- regulární – neterminál -> terminál ($S \to aS$, $S \to b$)
- bezkontextové – neterminál -> cokoliv ($S \to aSb$)
- kontextové – více ~~ntr~~ neterminálů na l-straně (počet se nemění: $ASB \to AAaBB$)
- rekurzivně vyčíslitelné – bez omezení
- kvalita gramatiky:
  - pokrytí – % vět jazyka L2 generovaných gramatikou ($|L_1 \cap L_2| / |L_2|$)
  - přesnost – % generovaných vět, které jsou správné věty jazyka L_2 ($|L_1 \cap L_2| / |L_1|$)

DC Gramatiky – rozšíření CFG, využívají rozdílových seznamů

- pravidla <hlava> --> <tělo> hlavu (neterminál) je možné přepsat na tělo (terminály)

Morfologická analýza – rozkládá ohebná slova na segmenty, určuje lemma

Test na shodu – argument Num (sg, pl) testuje shodu v čísle

Generativní síla DCG je větší než CFG

Princip kompozicionality: Význam složeného výrazu je funkcí významu jednotlivých podvýrazů, porušuje se kvůli významu (např. ustálená spojení)

Víceznačnost – lexikální, syntaktická, sémantická, referenční

Anafora – odkaz na dříve zmíněné objekty

Deixis – odkaz na údaje v jiných částech promluvy

PROLOG

UNIFIKACE – hledání nejobecnějšího unifikátoru dvou termů (sjednocování)

př.: info(Marcel, dana, Deti) = info(petr, dana, [jan, pavel])

po unifikaci:

Marcel = petr ; Deti = [jan, pavel]

BACKTRACKING – prohledávání do hloubky

REKURZE – opakované volání stejné f-ce

SYNTAX:

parent(tom, bob) – „tom" je rodičem „boba"

- **klauzule**: – seznam literálů
  - hlava :- tělo
  - 3 typy:
    - fakt – hlava bez těla p(X,Y) / p(X,Y) :- true
    - pravidlo – hlava i tělo p(Z,X) :- p(X,Y), p(Y,Z)
    - cíl – tělo bez hlavy ?- p(g,f) dotaz
- **Predikát**: seznam všech klauzulí se stejným funktorem (hlava) a aritou (počet termů) potomek/2
- **literál**: at. formule v negace
- **at. formule**: složený term
  - konstanta: nej malým písmenem a, 1, '!', [], se2, ...
  - proměnná: nej velkým písmenem v podtržítko A, _X, ...
- složený term: f(a, X) další objekty v P: datum(1, kveten, 1991)
  - funktor: datum
  - argumenty: 1, kveten, 1991 (arg/3 dává n-tý argument)
  - arita: počet argumentů (3)
  - identické termy = unifikované termy

PRÁCE SE SEZNAMY:

member(X, [X|_]) – true, pokud je prvek v seznamu

del(_, [], []) – smazání ze seznamu

insert(A, L, [A|L]) – vloží prvek postupně na všechny pozice

perm([], []) – vypíše všechny permutace seznamu

append([], L, L)

qsort([], []) – třídění seznamu

BIN STROMY

addleaf(nil, X, t(nil, X, nil))

delleaf(t(nil, X, Right), X, Right)